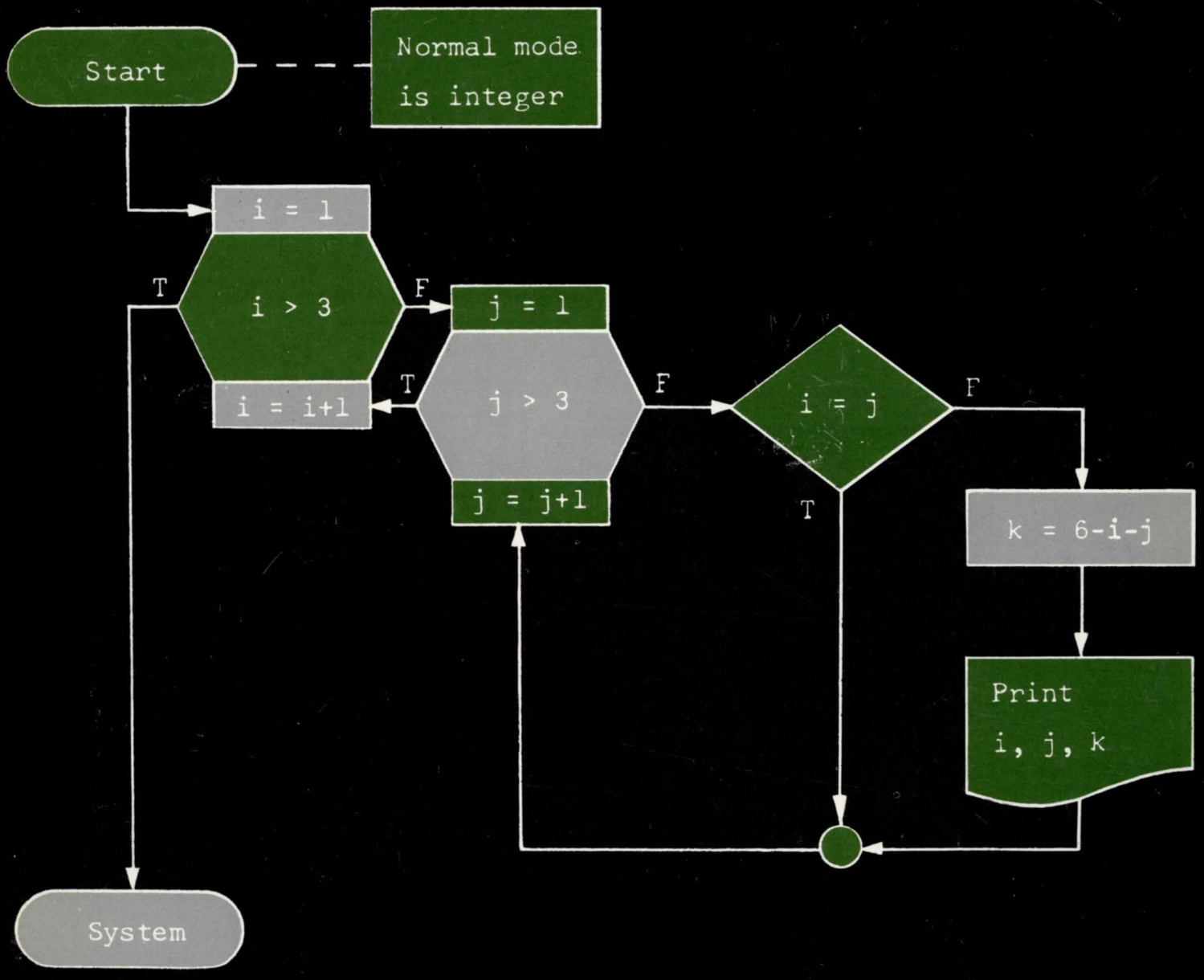
# An Introduction to ALGORITHMIC METHODS Using the MAD Language




Alan B. Marcovitz and Earl J. Schweppe

Marcovitz and Schweppe
ALGORITHMIC METHODS Using the MAD Language

Macmillan

# *An Introduction to Algorithmic Methods Using the MAD Language*

# AN INTRODUCTION TO ALGORITHMIC METHODS USING THE *MAD* LANGUAGE

ALAN B. MARCOVITZ
DEPARTMENT OF ELECTRICAL ENGINEERING
UNIVERSITY OF MARYLAND

and

EARL J. SCHWEPPE
COMPUTER SCIENCE CENTER
UNIVERSITY OF MARYLAND

THE MACMILLAN COMPANY · NEW YORK
COLLIER-MACMILLAN LIMITED · LONDON

© Copyright, The Macmillan Company, 1966

All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without permission in writing from the Publisher.

*First Printing*

Preliminary edition © copyright 1965
by Alan B. Marcovitz and Earl J. Schweppe.

*Library of Congress catalog card number: 66-27676*

THE MACMILLAN COMPANY, NEW YORK
COLLIER-MACMILLAN CANADA, LTD., TORONTO, ONTARIO

*Printed in the United States of America*

# PREFACE

This book has been written specifically as a lower division college text to introduce the student to those methods which are applicable to the implementation of algorithms on computing machines. Using a computer does not eliminate the need for careful work in the solution of problems; in fact, it places a premium on such work and adds some new aspects. In addition to the careful logical and mathematical analysis of problems, one must now consider the following: the structure of the available input data and the desired results; the structure and efficiency of the computational algorithm which describes the process of getting from the input to the results; the linguistic means of expressing this algorithm to the computer; the capacity and weakness of computers; the problems involved in debugging and running the algorithm on a computer; and finally, the careful verification of the results to assure that they are meaningful. This book is designed to present these aspects which are involved in effectively solving problems with the aid of a computer, and to relate them to other analytic techniques. The text assumes some basic background in mathematics, but most of the material is meaningful without the calculus.

A discussion of the characteristics of algorithms and an introduction to flowcharts for describing them are presented first. In order to allow almost immediate use of the computer, the basic features of the MAD language are also included in Chapter One. A description of the organization and properties of computers follows in Chapter Two. A metalanguage is described in Chapter Three so that the syntax of the MAD language may be formally defined throughout the text, rather than relying completely on definition by example. Using such a formal approach also helps prepare the student to learn additional computer languages. Numerous examples and flowcharts are included to aid in understanding the formal definitions. As the various features of the language are introduced, the new algorithm and data structures which they make available are illustrated with example problems. Many of the basic techniques of numerical methods are discussed and used for this purpose. The example problems are carefully analyzed with complete flowcharts and programs for their solution being included in many cases. The efficient use of the computer and the verification of the results are stressed throughout the text.

As indicated in the title, the MAD Language is used in this book to express the algorithms in a form acceptable for execution by a computer. MAD was developed at

the University of Michigan by Bruce W. Arden, Bernard A. Galler and Robert M. Graham and is described in a manual entitled, "The Michigan Algorithm Decoder", written by the developers. The design of the MAD Language was based partially on the first version of the International Algorithmic Language, sometimes called ALGOL 58, but the developers added numerous niceties of their own device. The language was first implemented on the IBM 704 and later converted for the IBM 7090, 7094. More recently it has been modified at City College, New York, under the guidance of Professor Ming L. Pei to operate on the IBM 7040, 7044. The MAD Language has been substantially used at City College, the University of Houston, the University of Maryland, the Massachusetts Institute of Technology, the University of Pittsburgh, and several other schools, in addition to the University of Michigan. Although not as sophisticated as the later versions of ALGOL or the newly emerging PL/I and not as widely available for use as FORTRAN, MAD has a number of attractions: the language reads easily and naturally, the decision making and other features of the language are quite general and powerful, the compiler is fast and informative in case of programming errors, the simplified input-output allows one to avoid the complexities of formatting on most student problems, there are good facilities provided in the systems for the automatic grading of student programs on the machine, and the keypunching of programs and data is well adapted to the equipment presently available in many installations. The prospect for the future is that MAD will be available--possibly in improved form--on both the GE 645 and the IBM System/360 Model 67.

The authors are grateful for the comments, criticism and assistance of many of their colleagues and students at the University of Maryland. Although it is not feasible to mention by name all those to whom credit is due, special recognition is given to Howard Tompkins, George Lindamood, Norman Glick and Edward Miller. Finally, the skillful typing and other assistance of Mary Schweppe has been of immeasurable help in the preparation of the manuscript.

Alan B. Marcovitz
Earl J. Schweppe

College Park, Maryland

CONTENTS

Chapter One ALGORITHMIC METHODS 1

1.1 Introduction 2

1.2 Algorithms and Their Implementation 7

1.3 Flowcharting 11

1.4 Elements of the MAD Language 16

1.5 A MAD Routine 23

1.6 Algorithms with Loops 26

Chapter Two THE COMPUTING MACHINE 35

2.1 Background 36

2.2 Organization 40

2.3 Binary Codes 44

2.4 Number Representation and Computer Arithmetic 49

2.5 Input and Output 60

2.6 Memory and Storage 63

2.7 Processing and Sequential Control 67

2.8 Software and the MAD Compiler 73

2.9 Using Computer Systems 81

Chapter Three DESCRIPTION OF LANGUAGE AND DATA 87

3.1 The Description of Languages 87

3.2 Structure of MAD Programs 93

3.3 Identifiers and Mode Declaration 96

3.4 Constants 100

3.5 Variables and Subscripts 106

3.6 Vectors and Initialization 109

3.7 Basic Input-Output 116

Chapter Four BASIC ARITHMETIC PROCESSES 127

4.1 Arithmetic Operations 127

4.2 Arithmetic Expressions 131

4.3 The Arithmetic Substitution Statement 139

4.4 Bitwise Operations 141

CONTENTS

Chapter Five DECISIONS AND TRANSFER OF CONTROL 145

5.1 Labeling of Statements 146

5.2 The Unconditional Transfer 148

5.3 The Variable Transfer or Switch 151

5.4 Boolean Mode and Relations 156

5.5 The Simple Conditional 160

5.6 Boolean Expressions and the Boolean Substitution Statement 168

5.7 The Compound Conditional 173

5.8 A Sample Routine for the Solution of Quadratic Equations 185

Chapter Six REPETITIVE PROCESSES 191

6.1 Repetition in Routines 192

6.2 Structure and Termination of Loops 195

6.3 The Iteration Statements 198

6.4 Iterative Solutions for the Zeros of Functions 217

6.5 The Newton-Raphson Iterative Method 232

6.6 Nested Loops 238

6.7 Approximating Areas under Curves 248

6.8 Approximating Slopes of Curves 256

Chapter Seven SUBROUTINES 261

7.1 The Use of Subroutines 263

7.2 Definition of External Subroutines 279

7.3 An External Function Definition 284

7.4 Other External Subroutine Definitions 290

7.5 Internal Subroutine Definitions 303

CONTENTS

Chapter Eight ARRAYS, MATRICES AND LINEAR EQUATIONS 309

8.1 Storage of Arrays 309

8.2 Arrays in MAD 312

8.3 Matrix Operations 319

8.4 The Gauss-Jordan Reduction - A Method of Solving Simultaneous Linear Equations 325

8.5 A Computer Algorithm for the Solution of Simultaneous Linear Equations 331

* * *

Bibliography 339

* * *

Appendix A - Abbreviations of Key Phrases in MAD 341

Appendix B - Number Systems 343

Appendix C - Debugging and Testing Techniques 361

Appendix D - Selected MAD Library Subroutines 367

Appendix E - Formatting and Input-Output 377

Appendix F - Other Features of the MAD Language 403

Appendix G - Syntactic Summary of the MAD Language 409

Appendix H - List of Complete Algorithms 427

* * *

Index 429

# *An Introduction to Algorithmic Methods Using the MAD Language*

Chapter One

# ALGORITHMIC METHODS

Although one may easily learn how to solve problems of many types, it is more difficult to recognize the common structure of problems which are seemingly quite different and to develop general methods for their solution. A precise description of how to solve a class of problems is called an algorithm. Thus it is appropriate to use algorithmic methods to denote the process of analyzing a class of problems in order to develop one or more algorithms for their solution. In analyzing a class of problems, a better appreciation of the structure of the individual problems in that class is usually gained and often the class itself can easily be extended.

Subject only to physical limitations such as size and speed, a computing machine can perform those processes which can be logically described as algorithms. With the recent advent and rapid growth in the use of computers, it is not then surprising that there is a corresponding interest in algorithms. Since computers, unlike humans, are usually unable to compensate for errors in their instructions, algorithms to be used on these machines must be accurate. Furthermore, the language used to express these algorithms must be one which is meaningful to both the man and the machine.

In this chapter, operations, algorithms, and computer programs will be introduced, along with some notations and languages for describing them. Some simple problems will be analyzed to illustrate and compare these various symbolisms. In the later chapters, increasingly complex problems will be studied and algorithms for their solution developed. The more powerful notation and language that will be needed to cope with these problems will gradually be developed and presented. Various structures for algorithms will be discussed and illustrated by example. Language features which make the expression of these algorithms easier will be introduced and used in the examples. The verification of the actual behavior of algorithms will be discussed and more information will be presented about the machines on which algorithms may be executed. Thus, although the explicit discussion of algorithmic methods occurs mainly in this chapter, the whole text is actually devoted to this subject and to the techniques necessary to implement algorithms on computing machines.

## 1.1 Introduction

There are four major steps in the development of an effective means for solving a class of problems on a computer. The first is the analysis of the problem class in order to formulate a logical problem model which can be adapted to processing on the computer. The second is the development of a practical and complete algorithm which, when applied to any one of the problems in the class, will produce the proper solution. The third is the precise expression of this algorithm in a language that the computer can accept as instructions. The fourth is verification of the algorithm. These steps proceed from an analysis of the class of problems to the creation, expression, and testing of an algorithm for its solution. Although parts of the first two of these steps must be taken to solve problems regardless of whether a computer is used, people often treat these steps more casually when solving problems by themselves than is possible when they are using a computer.

A person is usually interested in solving only a particular problem at any given time. He will probably do only enough analysis to cover the individual example at hand and then try various procedures he has learned until he finds one which yields a solution to the problem. He possibly checks this solution by some independent means and then considers his work complete. This way of solving problems is not adequate for the development of an algorithm to solve a class of problems on a computing machine. There is, however, a human activity which may be compared to that in which we are interested. This activity is the teaching of people to solve problems. In teaching, one must show the student how to recognize the various possibilities which may occur in the problem type under consideration and instruct him how to treat each of these situations. Thus the programming or instructing of a computer is to some extent similar to the training or instructing of people.

Just as with humans, computing machines are instructed to perform more complex operations in terms of simpler operations in which they have already been instructed or were initially built to perform. A newly designed and constructed computer can perform only certain very basic operations and the language in which it may be instructed is correspondingly primitive. The use of this language--usually called machine language--requires much detailed clerical work and takes considerable time to master. Using it alone would require so much time and attention that one could accomplish very little with the computer. For example, it might be necessary to

write

```
000 101 000 000 000 000 000 101 111 011 110 001
000 100 000 000 000 000 000 110 101 001 100 010
000 100 000 000 000 000 000 001 101 111 011 101
000 110 000 001 000 000 000 010 110 011 111 000
```

or

```
050000057361
040000065142
040000015735
060100026370
```

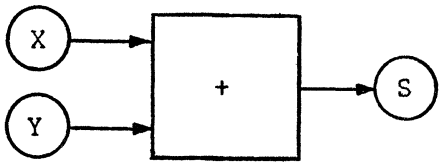
just to add three numbers.

To bridge the communications gap between man and machine, scores of intermediate languages have been developed. Each of these languages must be precisely defined for two reasons. First, the people using the language must know what may be written in it. Second, it must be possible to write a computer program to convert the intermediate language into machine language. Thus, the computer first translates instructions in these intermediate languages into its own language and only then can it carry them out or execute them. Some of these languages, such as FAP, CODAP, and AUTOCODER, have a close relationship with the machine language and their translators are usually called assemblers. More advanced languages, such as FORTRAN, MAD, ALGOL, COBOL, and PL/I, are oriented toward some type of problems and are easier for man to use. Translators for these, usually called compilers, must often perform quite complex transformations in order to produce machine language programs.

When properly instructed, a computer can accept instructions in any one of many different languages and behave in a more or less sophisticated manner. One would not hire a secretary who did not know how to read, type, and answer a telephone. Similarly, the user of a computer today expects that computer to be capable of doing certain routine things, including accepting instructions in several advanced languages. Furthermore, today's computer systems are expected to be able to detect and point out many of the errors which a user makes in giving his instructions. In this text, the MAD language will be used as an example of an advanced language and almost all of the algorithms will be expressed in it. MAD stands for the Michigan Algorithm Decoder and is an advanced language. It has many excellent characteristics both in simplicity of form and clarity of expression. It is relatively easy for students to learn and can be easily translated into basic machine language. Furthermore, the response of the MAD compiler to the user, especially in those cases where he makes an error, are more clear and helpful than is the case with many other compilers.

By using a language such as MAD to give instructions to a computer, many things can be assumed. For example, constants can be written as 1, -5, 37, 1.2, 6.943, or -8.67 and variables can be given names such as X, Y, or SAM. Not only can it be assumed that the computer can add, but also that it can be instructed to add two quantities called X and Y by writing X + Y. To give some variable S the value obtained by adding X and Y, one simply writes S = X + Y where we note that the equal sign has a different meaning than normally ascribed to it in mathematics. The meaning used here involves a change in the value of S rather than a statement of fact about the relationship between S, X, and Y. Similarly, subtraction, multiplication and division may be written as X - Y, X*Y and X/Y, respectively. The use of parentheses is also assumed to mean the usual grouping of quantities.

To make the basic operations in a machine more meaningful, diagrams like the following are sometimes helpful:




Here the diagram indicates that X and Y are sent to an adder in the machine and the resulting value is given to the variable S. Note that the X and Y stand for the inputs, the S represents the result and the + indicates the procedure used in getting from the first to the second. There certainly is an algorithm describing this process of addition, but it is built into the machine so that the user may simply assume it.

Several of these operations and variables may be combined to form an expression, such as

$$-2*3 + 6/(5-4)$$

which would, on evaluation, produce the result 0. Substituting variables for all the constants in this expression it becomes

$$-B*C + F/(E-D)$$

although it would certainly be possible to mix variables and constants in the same expression. Specifying the name A for a variable which is given the value obtained by evaluating this expression, the substitution statement

$$A = -B*C + F/(E-D)$$

is obtained. A diagram for this last statement is shown in Figure 1.1. Notice that if one disregards the contents of the dashed rectangle of this figure, the diagram represents the substitution statement as a whole.

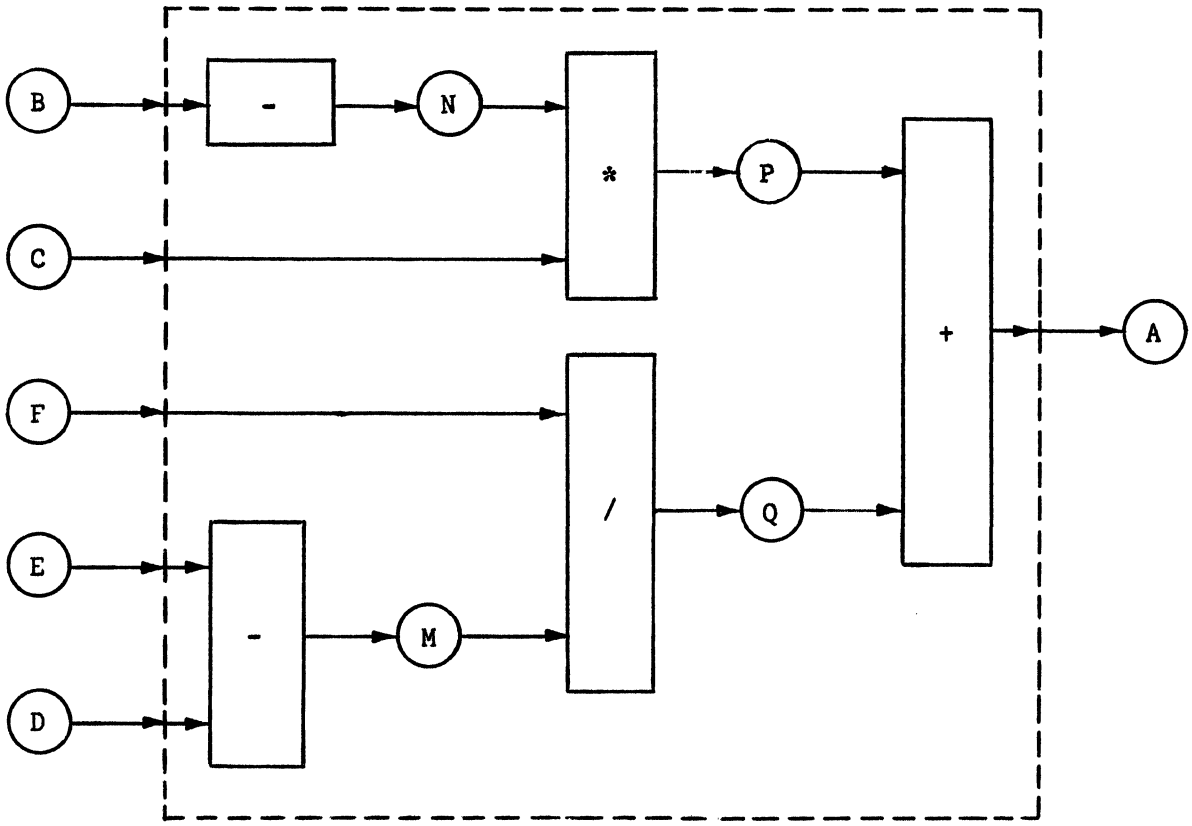



Figure 1.1 A diagram of A = -B*C + F/(E-D)

The variables M, N, P, Q shown in Figure 1.1 serve an intermediate role in the process of evaluating the expression. Using these intermediate variables it would be possible to replace the single statement above by five statements, namely:

```
N = -B
P = N * C
M = E - D
Q = F / M
A = P + Q
```

Thus, the expression actually represents a combination of more simple statements involving only single operations. In breaking down the above substitution statement into these simple operations it was necessary to give special treatment to the first minus sign. This sign represents negation, an operation applied to a single operand, and not subtraction. Since giving instructions to men or machines involves

combining known operations or procedures into new and more complex ones, this example illustrates the very essence of languages for these purposes.

In evaluating expressions, MAD follows the usual mathematical convention that multiplications and divisions are done before additions and subtractions. If two or more additions and/or subtractions are indicated, the left-most operation is performed first. Decisions concerning two or more multiplications and/or divisions are also resolved by doing the left-most operation first. The operation of negation (-) is done before those operations which involve two operands. Thus, the machine will treat

```
        a + b - c    as    (a + b) - c ,
        a / b * c    as    (a / b) * c ,
and     -a / b       as    (-a) / b    .
```

Many additional types of statements are necessary to write effective algorithms for complex problems, and more flexible ways of combining these statements must be available. In particular, it must be possible to control the order in which statements are executed and to make decisions as to which of alternative statements are to be executed. The details of these features are better left for later sections.

It is interesting to note at this point that the variable names and constants in the statements considered above serve the role of subjects and objects of the action. The operators themselves indicate the action taken and thus play the role of verbs. Thus, the languages developed for algorithms to be implemented on computers share some common structural features with natural languages such as English. The problems ahead of us involve learning not only to make correct declarations and statements in this language, but also the combination or composition of these "sentences" into larger units which are meaningful in content and function, as well as grammatically correct.

Exercises

1. Draw diagrams similar to Figure 1.1 for the following substitution statements:
   a) A = B*(C-D) + E/F*(-G)
   b) T = (R+S)/W - U*(-V)*S
   c) H = -U*V/M + (P-Q)*R
2. For each of the substitution statements of Exercise 1 find an equivalent set of simple substitution statements each of which involves a single operation.

## 1.2 Algorithms and Their Implementation

To make the definition of an algorithm quite precise, it would be necessary to use the concepts and symbolism of mathematical logic. Since knowledge of that field is not assumed on the part of the reader, it will be necessary to be less exact, but yet as accurate as possible.

The operands used in computations are normally numbers which are represented in some form, such as: written on paper, a set of holes punched in cards, or the positioning of wheels in a desk calculator. The representations of numbers, as well as letters, geometrical shapes and other symbols may also be subjected to manipulations such as ordering or alphabetizing, projecting, editing and changing the form of representation. Collections of these representations of numbers, letters and other symbols are called data. Particular items of data, especially those which play some special role such as the degree of a polynomial, are often called parameters.

An algorithm, as we will use it, is a precise specification of a computational or data manipulative process which will produce certain desired data as results within a finite amount of time when applied to particular input data which belong to the general class for which the algorithm was designed. Thus, algorithms are descriptions of transformations which can actually be carried out on data, and which are specified in terms of a combination of logical steps and/or decisions to be applied to the elements. There are three characteristic features of algorithms:

1) generality - the class of problems which the algorithm is designed to handle, usually expressed in terms of the domain of the input data.
2) conclusiveness - the requirement that the process described by the algorithm achieve the desired results after finitely many steps whenever admissible data are used as input.
3) definiteness - the precision of the specification which leaves no place for question or interpretation and is universally comprehensible.

The above features are discussed in a book by A. A. Markov[1] where a precise treatment of the definition of an algorithm may be found. Note that generality has mainly to do with the class of data representing the specific problems which the

[1] A. A. Markov, Theory of Algorithms, translated from Russian, Office of Technical Services, Washington, D.C., 1961.

algorithm is designed to handle. Conclusiveness has to do with the orientation of the algorithm to reach a stopping point and thus produce some results. Definiteness primarily pertains to the process of getting from the input data to the result. Thus, these three points concern input, output and processing, respectively.

In a looser sense, examples of algorithms abound in all our lives. The instructions for completing an income tax return, the steps of going through a cafeteria line, the directions as to how to drive from one point to another, and the procedure inherent in an assembly line for building a product such as an automobile, are but a few examples. Usually, none of these "algorithms" is described precisely and the processes involved function only because of frequent human intervention and interpretation to keep them on the desired course. In working with a computer, or in automating other processes with the help of computers and other automata, it is necessary to more completely understand and specify what is to be done.

In mathematics the well known Euclidean Algorithm specifies a process whereby the greatest common divisor of two natural numbers may be found. The two natural numbers constitute the data in this case and the greatest common divisor is the result of the procedure. Actually the child in school is taught many other algorithms which are simply not advertised as such. For example, the process of adding two numbers, such as

$$\begin{array}{r} 21 \\ +16 \end{array} \quad \text{or} \quad \begin{array}{r} 1342 \\ +784 \end{array} \quad \text{or} \quad \begin{array}{r} 159683297 \\ +43509682 \end{array} ,$$

can be described as an algorithm based on the ability to add three digits, the two in any column and a carry from the column to the right. At home and in the schools the processes of counting and adding digits are established in children mainly by rote means, but without these it would not be possible to teach the child to add numbers involving several digits. This brings out another feature of algorithms. The algorithm must always be expressed in terms of certain fundamental operations or elementary procedures which it is assumed are meaningful to the people and/or machines which will execute or otherwise use them.

In developing algorithms for computers, it is usually desirable to generalize as much as possible the class of problems which it will solve without unduly complicating the work. For example, it is almost as easy to develop an algorithm to add n numbers where $0 \leq n \leq 10000$ as it is to develop one to add exactly twenty numbers. Having chosen the class of problems to be solved, one must reason out all possible circumstances in this class which affect the process of solution. Then one must find efficient ways of producing the solutions in each case and make the instructions quite definite as to how to handle each of these possibilities. The

definiteness requires that the execution of an algorithm always begins with the same instruction and makes its own decisions as to which case it is treating. These cases may be indicated explicitly in the data, although usually they are only implicitly contained in it. Finally one must make sure that the computer can indeed achieve the required solution in a reasonable amount of time and especially that it not be caught in some logical quandary which causes it to be unable ever to conclude its work. In addition to treating all possible sets of admissible data, it is usually desirable for the algorithm to reject data which are not admissible and give some intelligible response concerning the reason for rejection. Even though the analysis of the problem and the development of an algorithm may have been carefully done, it will be found necessary to test algorithms on the computer and verify that they do what was intended under all possible circumstances. This testing is very important and requires careful selection of the data used in the verification of the algorithm. In most cases it is possible to limit these tests to fairly small amounts of time if they are properly designed.

As a quite simple example which will be traced through much of this chapter, consider the problem of finding all values of x such that

$$ax + b = 0$$

where a and b are any real numbers.

In only two steps almost any high school student can immediately describe how to "solve" this type of problem completely, specifically:

1) Subtract b from both sides of the equation.
2) Divide both sides by a.

Thus, they might try to use the mathematical solution $x = -(b/a)$ to produce solutions in all cases. Unfortunately, this solution is not valid in case $a = 0$ and in fact two separate cases must be considered in that eventuality. Furthermore, an algorithm designed for a computer must specify where the data is to be obtained and what is to be done with the results.

A description of an algorithm for the solution of this problem is contained in the following instructions:

0) Read a set of data (a and b).
1) If $a \neq 0$, do 7 next. Otherwise do the next instruction.
2) If $b = 0$, do 5 next. Otherwise do the next instruction.
3) Print a comment that no value of x satisfies this equation.
4) Terminate.
5) Print a comment that any value of x satisfies this equation.
6) Terminate.
7) Calculate $x = -(b/a)$.
8) Print the result x.
9) Terminate.

It is assumed in this list of instructions that the next instruction always follows unless otherwise stated. The instruction numbered 0 is always taken as the initial instruction. The process may stop on any of the instructions 4, 6 or 9. It terminates after a finite number of steps, in this case exactly five steps for each possible path. This description is complete and the algorithm works for all possible values of the input data. Thus, it does fit the description of an algorithm.

In other cases the number of steps may be dependent on the path taken and, indeed, may even vary for the same path. As will be seen in Section 1.6, most algorithms involve some steps being done more than once and so they "loop" through certain steps. In these cases the number of times through the loop and hence the number of steps may vary with the data. Even under these circumstances the algorithm can be conclusive by terminating after a finite but not a fixed number of steps. Although an algorithm must terminate, it is not usually economical to let a computing machine actually stop. Computers, like people on the job, usually cost the same money whether working or loafing. Fortunately, they have not yet formed unions to demand coffee breaks or whatever type of breaks might be appropriate. Computers do, however, break down and require either major or minor repairs to put them back in working condition. Since a computer program is nearly always run under the supervision of a master programming system, the most frequent termination is a return to that system. This return may be thought of by the user as an instruction to the computer to "do something else useful".

Although the above algorithm is very simple, it illustrates many of the properties of algorithms. Note that even here it is assumed that division is a known operation and that decisions can be made on a basis of the relationship of variables and numbers. The description of the algorithm given above is fairly precise, but we will find it possible to simplify it somewhat and make it even more explicit by use of the MAD language. The MAD description will, however, be very similar to the one above. Before introducing the MAD language and writing this algorithm in it, another notation for expressing algorithms will be introduced. This notation, consisting of flowcharts, is graphical in nature and better illustrates the paths taken in executing the algorithm.

Before going on to other material, there are a number of terms and concepts related to the implementation of algorithms on a computer which should be introduced, discussed and related. As has been indicated earlier, algorithms are specified in terms of statements calling for known processes such as addition and division. If one looks more closely at each of these processes, there is an algorithm which

specifies how it is carried out. More complex processes are defined in terms of those processes which are already known to the system. Most programming systems already contain the definitions of processes for finding square roots, computing trigonometric sines, taking logarithms and the like. To use these processes one does not need to understand the algorithm which is used, but needs only know how to furnish the correct inputs and how the results are related to these inputs.

The term program is used herein to denote a complete set of statements in a particular language which specifies the process of solving a class of problems. In order to be considered a computer program, the language used must be acceptable to one or more computer systems which are capable of carrying out the solution process when given this program and the necessary data. Since our primary interest is in computer programs, we often use "program" to mean "computer program." Sometimes programs are also called routines. It is often quite natural to break programs up into smaller parts or subprograms which are controlled or used by a main program. We will prefer to use the terms subroutine and main routine for these subsections of a program. All of these concepts will be discussed more completely at a later point.

## 1.3 Flowcharting

In this section we introduce flowcharts, a graphical notation for algorithms. Although these representations are not in some ways as precise as the languages we will actually use to write algorithms for computers, they are quite useful in the analysis and development of the algorithm. They are also useful in the documentation of an algorithm so that others may understand the various paths through the steps of the algorithm which may occur during its execution on the computer. They also display quite clearly the decision points and the various branches which may be taken from these points.

A flowchart consists of boxes which contain notation representing processing, input-output, operations, decisions, or related information, and lines which connect these boxes. Although the shape of the boxes used is not vital to the idea of a flowchart, it is helpful to use different shaped boxes, such as rectangles, hexagons, circles, and diamonds, for different processes or functions. The connecting lines may be of two types. Flow lines are drawn solid with an arrowhead at one end representing the direction of flow. They indicate the paths which may be taken through the algorithm from one processing or other box to another. Annotation lines are drawn dashed and involve no direction. They simply attach a box containing certain remarks or declarations to a point in the flowchart where they will make the process more clear. Most boxes of a flowchart may have just one flow line

entering them. This entry point is indicated by placing the arrow head so that it touches the box, pointing toward it. These boxes may also have exit points, which are indicated by the line of flow simply connecting to the box. These exit points are often labeled to indicate the conditions under which that flow line is taken.

The flowchart of the algorithm introduced in the previous section for the solution of linear equations of the form $ax + b = 0$ is shown in Figure 1.2. This

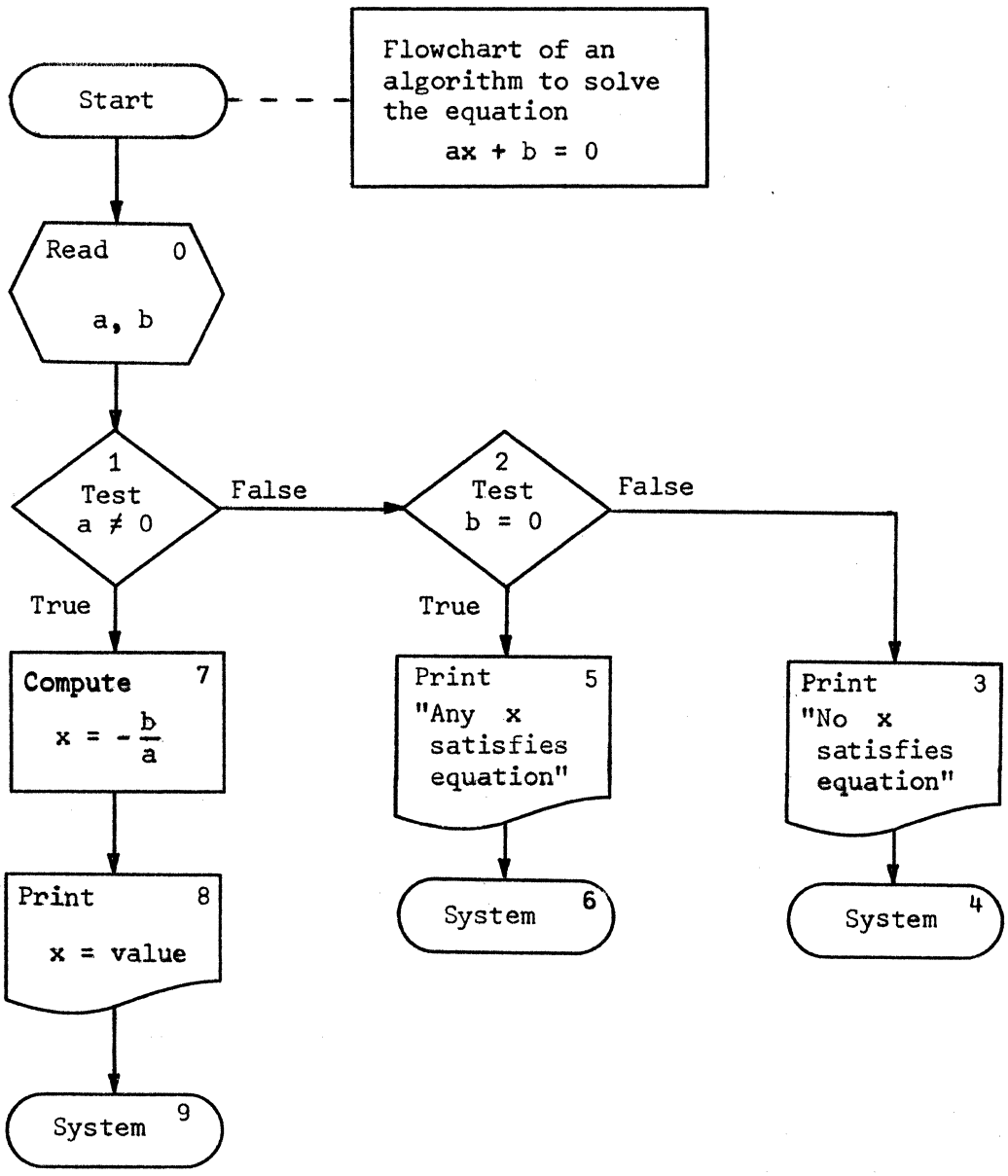
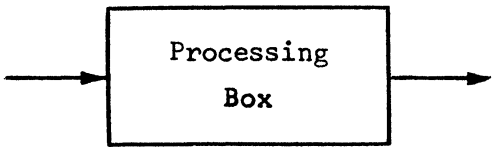



Figure 1.2 A simple flowchart

flowchart illustrates many of the features of these charts. The order of execution of the steps in the algorithm can easily be traced by proceeding consecutively from one box to the next (following the flow lines in the direction of the arrows). Some boxes have more than one exit and represent decision points in the algorithm. The logical basis (true or false) for the various exits from these boxes is shown where the lines emerge. The step numbers from the description of the algorithm in the previous section are also shown in the boxes.

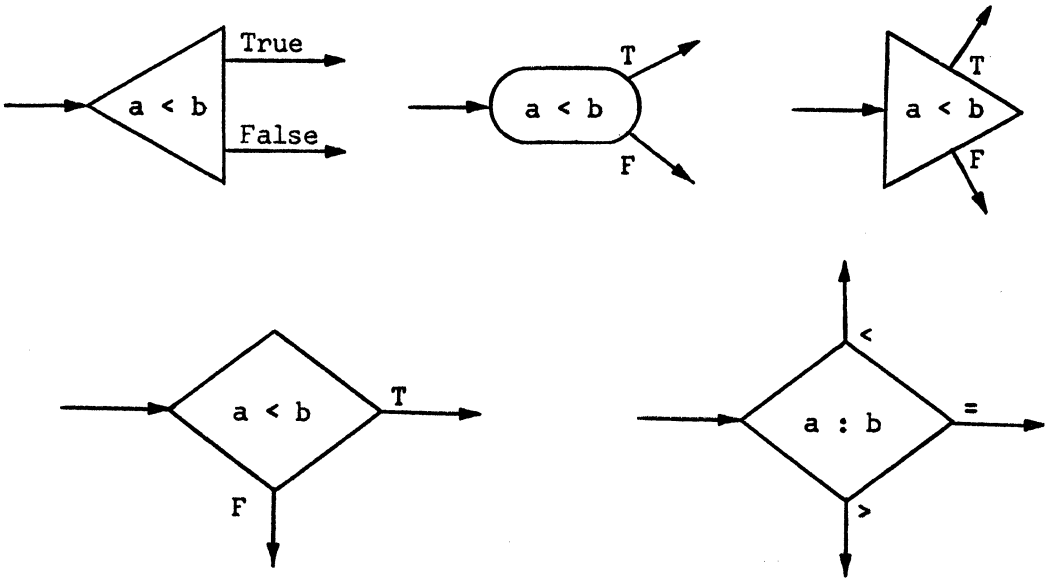
Several shapes of boxes are illustrated in the above example. We will now discuss in detail these boxes and some others individually.

A processing, substitution, or operation box is usually drawn as a rectangle and represents one or more steps which are executed always in the same order, with




no decisions or alternative paths allowed. These boxes must have a unique entry point and a unique exit point.

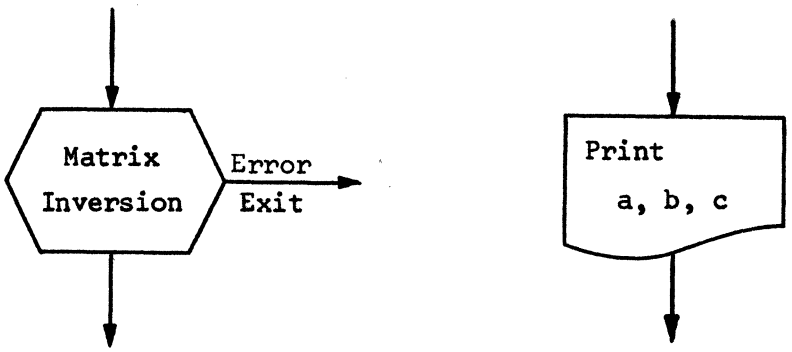
A decision box is usually drawn in a diamond shape, but triangles and ovals are also used for this purpose. These boxes usually have a unique entry point;




however, sometimes two entry points will be used in order to make the flow lines fall more conveniently. A relationship between two variables or more complex expressions may be placed in the box with True and False labeling the two exit points

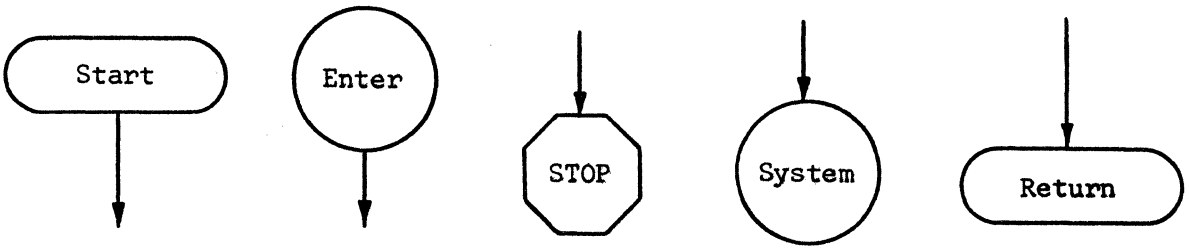
and indicating the conditions under which these exits will be taken. Another notation is sometimes used which separates two variables by a colon in the box and uses the relations between the variables to label the exit points. The diamond is more flexible, and is widely accepted as a standard. It easily allows either a two way or a three way branch and in the case of two exits a choice of directions may be made to better arrange the flowchart on the page.

A predefined process or subroutine box is drawn as a hexagonal or special shaped figure and represents a group of operations which are not detailed on the particular flowchart being used, for example, a library subroutine. These boxes may have multiple entry and exit points.




We will use the standard procedure box for reading data into the machine, but a special box signifying a torn off sheet of paper for printing output.

An origination or termination box is usually drawn as an oval or a circle. These boxes indicate the beginning, start, or entry of an algorithm and the end, finish, or exit of an algorithm, respectively. An origination box has exactly one exit line and a termination box has exactly one entry point.

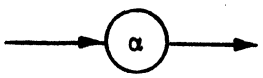



In the case of subroutines, the termination may be a return to the program which called it. Some people, out of consideration for the highway department, like to use octagonal boxes when the computer is really to be stopped.

Sometimes it is difficult to show all the lines of flow without their crossing or causing confusion; furthermore, a flowchart may often extend over more than one page. Thus, it is necessary to have some means of indicating the connection of

lines of flow which do not actually meet. A connector box drawn as a small circle is useful for this purpose. Thus,

actually means




or even



removing the connector.

A letter or symbol is used to label the connectors and thus link them. There must be exactly one such circle with a given label and a flow line exiting from it, but many such circles may indicate connection to this one. These connector circles may also be used as merge points in the flowchart allowing several lines of flow to

join. In this case a label is not necessary, but is sometimes useful.

Although it is not necessary for these connectors to all be on the same page, a special symbol is recommended by some when these connectors designate exit from or entry to a page. Thus,

might be used in this case.

An annotation box may be used to make declarations, supplementary remarks or explanations which will clarify the algorithm at any point in the flow. These boxes are rectangular and sometimes have a dashed line as one side.

Two types of flowcharting boxes will not be introduced until later in the text. One involves a variable connection or switch and will be discussed in Chapter 5. The other, which represents the iteration statement of the MAD language, will be developed in Chapter 6.

The boxes of a flowchart may be numbered or labeled to tie them in with the program which is written to correspond with them. The notation which may be used within the boxes is not rigidly specified; however, the use of English and mathematical notation is generally preferred, rather than computer-oriented languages.

## 1.4 Elements of the MAD Language

In order that the reader may begin testing some algorithms immediately, we will now introduce a portion of the MAD language. Since features of MAD that are presented here will be described only in sufficient detail that they may be used in simple problems, omission is not to be understood as implying illegality. A complete discussion of the rules of the MAD language begins in Chapter 3.

The characters with which the MAD language is written are:

1) Upper case letters of the alphabet (i.e. A,B,C,...,Z)[2]
2) The ten digits (i.e. 0,1,...,9)
3) Eleven special characters, namely + - * / , . = ' $ ( )
4) The blank.

These characters are used to represent numbers, to form names of various items and to write statements or declarations about these numbers and named items. These statements and declarations are usually punched on cards (one per card), beginning no earlier than column 12 and ending no later than column 72. Statements may be labeled, i.e. given a name, so that they may be referred to elsewhere. These statement labels are punched in the first ten columns of the card. Blanks are usually ignored.

Numeric constants may be in one of two modes, integer or floating point. Integers may be positive, negative, or zero and are written without a decimal point (e.g. 0, 365, -1, 27, +27, -1641). The magnitude of integers should be less than approximately $10^8$. Floating point mode numbers may be written as decimal fractions with a decimal point (e.g. 1., 3.641, -.000000000000174, 163421.). They may also be written as an integer or decimal fraction followed by an E and an integer. The E is interpreted as "times 10 to the power" and the integer as the power of ten. Thus 1.3E-6 is $1.3 \times 10^{-6}$ or .0000013, and 10000. may be written 1E4.

Names of variables and labels of statements begin with a letter and consist of up to six letters or digits (no special characters). Examples are X, BØY,[2] Y2, and A43PQ2. Any name may be used either as a variable or a label, but not both.

All variables are assumed to be in floating point mode, i.e. the value of these variables are floating point mode numbers, unless otherwise declared. If a variable is to be in integer mode, the name of the variable would appear in a declaration of the form

```
INTEGER I,J,PQ3
```

[2]The letter "O" is often slashed to distinguish it from the digit zero "0".

which declares that I,J and PQ3 are the names of integer mode variables.

A non-negative integer mode constant, variable, or arithmetic expression may be used as the subscript of a variable. A set of variables with a common name and a single subscript is often referred to as a vector or a linear array. Subscripts are enclosed in parentheses, e.g. A(1), X(I), P(Q+1), RS(0). A zero subscript (and the parentheses) may be omitted, i.e. RS(0) and RS have the same meaning. All elements of a vector (i.e. all variables with the same name) must be in the same mode. Space may be reserved for a vector, using a declaration of the form

```
DIMENSIØN  X(20), A(1)
```

where the constant enclosed in parentheses is the largest subscript to be used. Thus the above declaration causes space to be reserved for 21 components of X, namely X(0) through X(20), and two components of A, namely A(0) and A(1).

The last statement of a MAD main routine is always

```
END ØF PRØGRAM
```

This statement serves as a signal both during the translation and execution of the routine that there are no further statements or declarations. Declarations may occur anywhere in a routine before the END ØF PRØGRAM statement. The execution of the END ØF PRØGRAM statement causes a return to the system.

A statement for data input is

```
READ AND PRINT DATA
```

which reads the next set of data from cards and prints an image of the cards read. If no data remains, this statement results in the return of control to the system and execution of the next program.

An item of data is of the form A = 6 or X(7) = 14.2, i.e. variable, "equal sign", constant, or Y(1) = 4, 3, 6, i.e. variable, "equal sign", list of constants. In the first case, the variable is set equal to the constant, i.e. A is made equal to 6. In the latter case, the variable, Y(1), is set equal to the first constant, the variable with the same name but one larger subscript, i.e. Y(2), is set equal to the second constant 3, and so forth. A data set is a list of data items separated by commas and terminated by an asterisk. It may be punched anywhere in the first 72 columns of the card. A typical set of data might be

```
A = 3.64, B = 1.6E-13, I = 1.2, 2.3, 4.1, J = 1 *
```

Before the data is stored, it is converted to the mode of the variable if necessary.

A statement for the printing of results is of the form

```
PRINT RESULTS X, Y, Y(1), I, J, A(3)...A(15)
```

where the values of the variables X, Y, Y(1), I, and J are printed, four to a line,

in a format (condensed here) such as

```
X = 1.364211E-01,      Y = -2.643502,    Y(1) = 4.617205,      I =          5
J =            7
```

and blocks of results, i.e. consecutive elements of a vector, are printed eight to a line, in a format such as

```
      A(3)...A(15)
 763      -421         0         0        31        12         8       647
  21         2        43       -21     -4162
```

Comments may be printed with a statement of the form

```
PRINT CØMMENT $ ANYTHING GØES.$
```

where everything between the dollar signs except the first character after the first dollar sign is printed. This first character is used to control the movement of the carriage from line to line. The carriage control character may be a blank, in which case the comment will be printed on the next line, a zero (0), in which case a line will be skipped before printing, or a (1), in which case printing begins at the top of a new page. (Other carriage control characters will be discussed in Chapter 3.) Blanks are not ignored between the dollar signs.

The common arithmetic operators are shown in Table 1.1. Parentheses are used in the same manner as in algebraic notation. Redundant parentheses may be used. All arithmetic operations are performed in the mode of the operands if both operands have the same mode and in floating point mode otherwise.

Table 1.1

| Operator | Operation | MAD Expression | Algebraic Expression |
|---|---|---|---|
| + | addition | A + B | a + b |
| - | negation | -A | -a |
| - | subtraction | A - B | a - b |
| * | multiplication | A * B | ab |
| / | division | A / B | $\frac{a}{b}$ |
| .P. | exponentiation | A .P. B | $a^b$ |
| .ABS. | absolute value | .ABS. A | $\|a\|$ |

An arithmetic expression consists of variables and constants operated on by arithmetic operators. Detailed rules for the construction of arithmetic expressions are discussed in Chapter 4; however, they follow closely the normal rules of algebra. The order in which expressions are evaluated is governed by the rules of precedence; with all operations of higher precedence performed before those of lower precedence. Operations of equal precedence are performed from left to right. In any case, operations within parentheses are performed before those outside those parentheses. Table 1.2 indicates a partial list of precedence. Examples of arithmetic expressions are

```
A + B
P * Q - C * D
(P * Q) - (C * D)
.ABS. (A + B) - 3. .P. I
```

Note that the second and third expressions produce identical results. Also, if C and D are both in integer mode, their product is calculated in integer mode independent of the modes of P and Q. The mode of an arithmetic expression is the mode of the result which is floating point if any of the variables or constants are floating point mode and integer mode otherwise.

Table 1.2

```
.ABS.
.P.
-  (negation)
*     /
+     -  (subtraction)
```

A substitution statement is of the form

```
X = A + B
```

i.e. variable (possibly subscripted), "equal sign", arithmetic expression (possibly just a constant or a variable). The operation of the statement is as follows:

1) The expression is evaluated.
2) The value is converted to the mode of the variable on the left.
3) That variable is set equal to the value of the expression.

For example, the two consecutive statements

```
I = 5
I = I + 1
```

where I is in integer mode produce the following results:

1) I is set equal to 5, the value of the expression to the right of the first equal sign.
2) The expression I + 1 is evaluated, using the current value of I, namely 5, and ignoring the variable to the left of the equal sign.
3) I is then set equal to the value of that expression, namely 6.

Whenever control is to be transferred to another portion of a routine, i. e. statements are not to be executed in order, the statement to which control is transferred must be labeled.

The transfer statement is of the form

```
TRANSFER TØ S
```

where S is the label of the next statement to be executed.

Many algorithms, including the example discussed in Sections 1.1 and 1.2, involve decisions and branches based on those decisions. Decisions in MAD may be based on the value of relations, either "true" or "false". A list of the relational operators is given in Table 1.3. A relation consists of two arithmetic expressions

Table 1.3

| Operator | Operation | MAD Expression | Algebraic Expression |
|---|---|---|---|
| .G. | greater than | A .G. B | a > b |
| .L. | less than | A .L. B | a < b |
| .GE. | greater than or equal to | A .GE. B | a $\geq$ b |
| .LE. | less than or equal to | A .LE. B | a $\leq$ b |
| .E. | equal to | A .E. B | a = b |
| .NE. | not equal to | A .NE. B | a $\neq$ b |

connected by a relational operator. Examples of relations are

```
4 .G. 5
```

which always has the value false and

```
B .P. 2 - 4.*A*C .GE. 0.
```

which is true when the value of the expression $b^2 - 4ac$ is greater than or equal to zero and false otherwise.

The simple conditional statement may be of the form

```
WHENEVER A .G. B, TRANSFER TØ S
```

or

```
WHENEVER I .L. 0, I = -I
```

In each case, the word WHENEVER is followed by a relation, a comma, and another statement. This latter statement (used as part of the simple conditional statement) may be a transfer statement, an input or output statement or a substitution statement, but not another conditional statement or END ØF PRØGRAM. The operation of this statement is as follows:

1) The relation is evaluated.
2) If its value is false, the statement following the comma is ignored and the next statement after the conditional one is executed, i.e. the statement on the next line.
3) If the value of the relation is true, the statement after the comma is executed. If that statement is a transfer statement, execution continues at the point to which control was transferred. Otherwise, the next statement is then executed.

The MAD library contains a large number of mathematical functions in the form of subroutines that may be used by making the appropriate call. We will describe the method of calling some single-valued functions of a single parameter, a list of which is given in Table 1.4. All functions described here require floating point mode parameters and produce floating point mode results. The function call consists of the function designation (which always ends with a period) following by a floating point mode constant, variable, or expression enclosed in parentheses. A function call of this type may be used in an arithmetic expression in the same way that a variable is used. Examples of MAD expressions containing function calls and the equivlaent algebraic expressions are

| | |
|---|---|
| `SQRT.(B.P.2 - 4.*A*C)` | $\sqrt{b^2 - 4ac}$ |
| `4. * SIN.(X) - X*X` | $4 \sin x - x^2$ |

Table 1.4

| Function Name | MAD Call | Mathematical Expression | Notes |
|---|---|---|---|
| SQRT. | SQRT. (X) | $\sqrt{x}$ | $x \geq 0$, $\sqrt{x} \geq 0$ |
| SIN. | SIN. (X) | sin x | x in radians |
| CØS. | CØS. (X) | cos x | x in radians |
| EXP. | EXP. (X) | $e^x$ | $x < 87.3$ |
| ELØG. | ELØG. (X) | ln x | $x > 0$ |
| ARCCØS. | ARCCØS. (X) | $\cos^{-1} x$ | $0 \leq x \leq 1$<br>$0 \leq \cos^{-1} x < \pi$ |
| ARCSIN. | ARCSIN. (X) | $\sin^{-1} x$ | $0 \leq x \leq 1$<br>$-\pi/2 \leq \sin^{-1} x < \pi/2$ |

Exercises

1. Draw a flowchart corresponding to each of the following sets of MAD statements:

a)

```
    WHENEVER A .G. 1, A = 1
    B = 0
```

b)

```
    WHENEVER Q .L. J, TRANSFER TØ S
    A = A + 1
S   B = 0
```

c)

```
    WHENEVER Q .GE. J, A = A + 1
    B = 0
```

2. Write MAD expressions for each of the following algebraic expressions:

a) $3x^2 - 4x + 1$

b) $4c^x \sin y - x^3$

c) $\frac{a+b}{2cd}$

d) $|a|c||$

e) $(-2)^x$

## 1.5 A MAD Routine

Routines in MAD, as in other existing computer languages, must be linear, i.e. there must be a single sequence of statements. Flowcharts, on the other hand generally have branches and two or more sets of parallel steps. Thus, it is often necessary to modify the flowchart (at least make the modification mentally) to produce a single line of blocks. There are usually many ways of modifying the flowcharts, particularly if we modify some of the relations (note that A.NE.B is true whenever A.E.B is false, and false whenever A.E.B is true). One modification of the flowchart of Figure 1.2 to exhibit this property is shown in Figure 1.3a and the corresponding MAD coding is given in Figure 1.4a.

The first line in the code (and any other line with an R in column 11) is a remark. In this case the remark serves as a title for the routine, but they may also be used to give other information about the routine to the user. Remarks are ignored by the translator, but are printed with the routine whenever the routine is listed. They have no effect on the operation of the program.

The first statement in the routine, READ AND PRINT DATA, takes a single set of data from cards and stores it in memory. Since this routine does not return to this (or any other) input statement, it can handle only one set of data. The MAD compiler ignores everything on the card beyond the phrase READ AND PRINT DATA, and thus the names of the data to be read are often added to that line. This augments the remarks in the annotation of the routine.

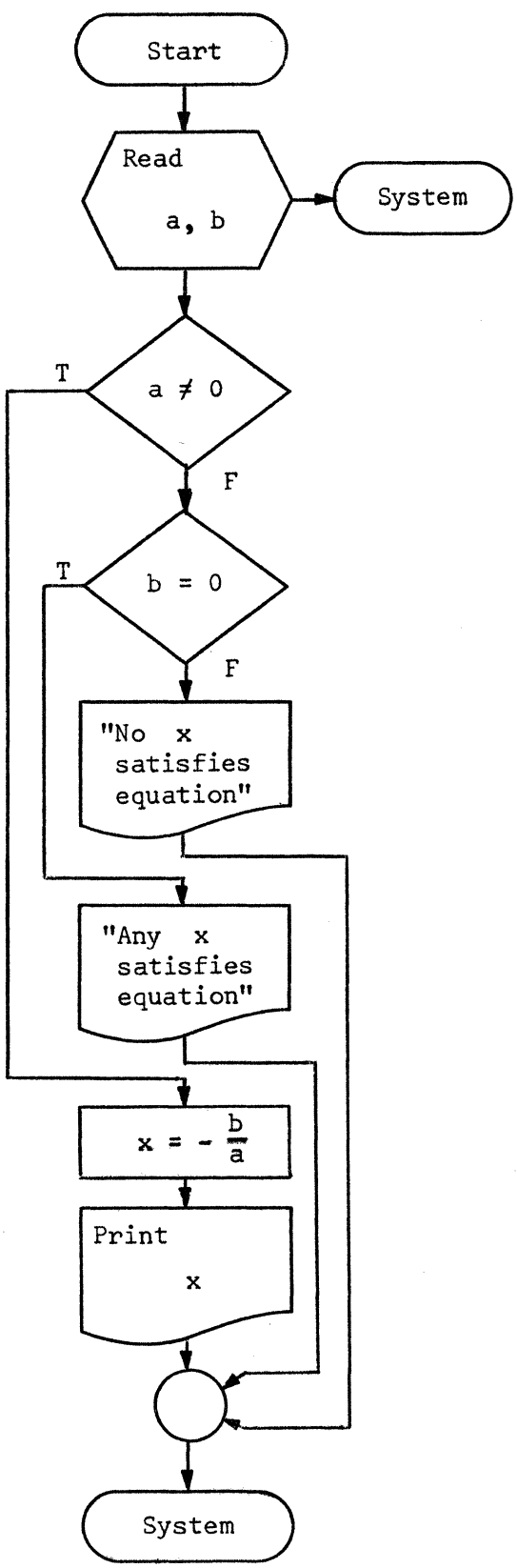
After the data is read, the relation A.NE.0. is evaluated. If the value is true, then the statement labeled S is executed. The results are then printed, END ØF PRØGRAM is executed, and control is returned to the system. A typical printout for this case might be

```
A = 2.4, B = 3.6*
        X = -1.500000
```

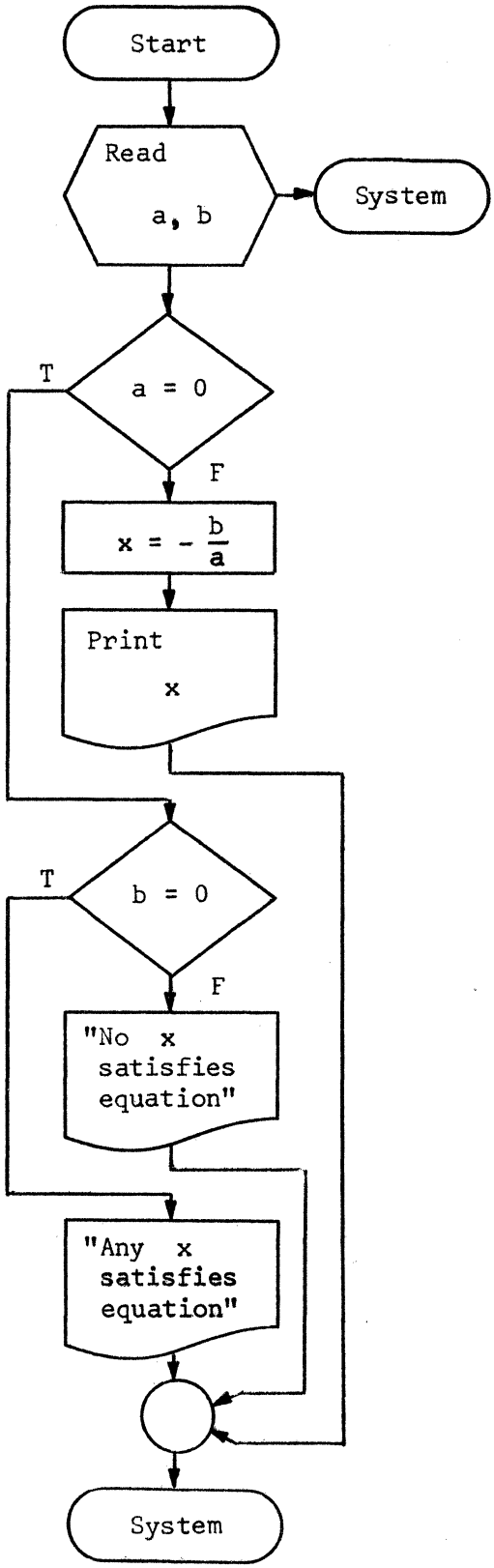
where the first line is an image of the data card and the second line results from the statement PRINT RESULTS X.

If the value of A .NE. 0. is false, the second relation, B .E. 0. , is evaluated. If it is false, the first comment is printed and control is transferred to the statement labeled U, END ØF PRØGRAM. Otherwise, control is transferred to the statement labeled T, the second comment is printed, and END ØF PRØGRAM is executed.

The linearization suggested in the flowchart of Figure 1.3a is only one of many. A second modification is shown in Figure 1.3b, with the corresponding MAD routine in Figure 1.4b. Various linearizations of a flowchart should be considered


Start
Read
a, b
System
T
a ≠ 0
F
T
b = 0
F
"No x satisfies equation"
"Any x satisfies equation"
x = - b/a
Print
x
System

(a)


Start
Read
a, b
System
T
a = 0
F
x = - b/a
Print
x
T
b = 0
F
"No x satisfies equation"
"Any x satisfies equation"
System

(b)

Figure 1.3

so that the resulting routine is most readable and logically simple.

```
          R   RØUTINE FØR SØLUTIØN ØF A * X + B = 0
           READ AND PRINT DATA      A,B
           WHENEVER A .NE. 0., TRANSFER TØ S
           WHENEVER B .E. 0., TRANSFER TØ T
           PRINT CØMMENT $ NØ VALUE ØF X SATISFIES THIS EQUATIØN. $
           TRANSFER TØ U
T          PRINT CØMMENT $ ANY VALUE ØF X SATISFIES THIS EQUATIØN. $
           TRANSFER TØ U
S          X = -B/A
           PRINT RESULTS X
U          END ØF PRØGRAM
```

Figure 1.4a

```
          R   RØUTINE FØR SØLUTIØN ØF A * X + B = 0
           READ AND PRINT DATA      A,B
           WHENEVER A .E. 0., TRANSFER TØ TESTB
           X = -B/A
           PRINT RESULTS X
           TRANSFER TØ END
TESTB      WHENEVER B .E. 0., TRANSFER TØ ANYX
           PRINT CØMMENT $ NØ VALUE ØF X SATISFIES THIS EQUATIØN. $
           TRANSFER TØ END
ANYX       PRINT CØMMENT $ ANY VALUE ØF X SATISFIES THIS EQUATIØN. $
END        END ØF PRØGRAM
```

Figure 1.4b

## 1.6 Algorithms with Loops

As another example of an algorithm, we will consider the following problem:

> Calculate the sum of a set of n numbers, $a_0$, $a_1$, ..., $a_{n-1}$, where $0 \leq n \leq 100$. The integer n and the floating point mode numbers $a_i$ are to be read as data and the sum is to be printed out. If n is zero, there are no numbers to be added and the sum is zero. After each calculation, another set of data is to be read and the calculations repeated for that data.

A flowchart that indicates the basic structure of this algorithm is shown in Figure 1.5, where the details of the calculation box will be shown below. This structure

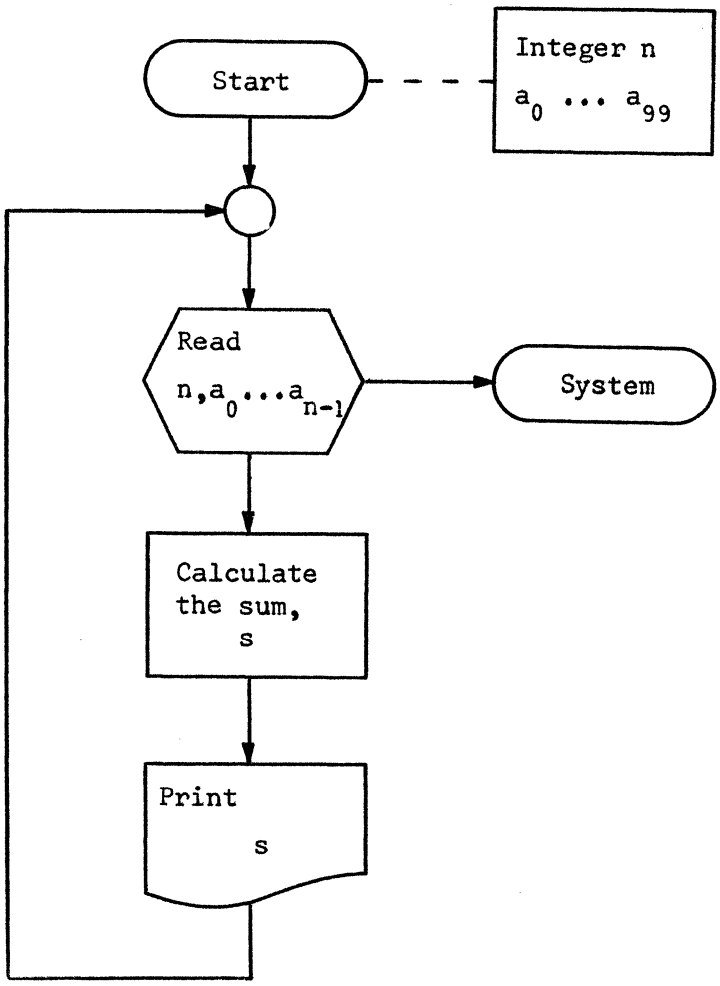



Figure 1.5

is similar to that of many algorithms, in that it is designed for the processing of as many sets of data as are available. Unlike the earlier example, this algorithm contains a loop, i.e. a portion of the algorithm is repeated (the whole algorithm in this case). The loop is terminated and control is returned to the system when the read step is unable to find any more data.

Before considering the calculations necessary for this algorithm, let us consider those for a more simple problem, namely the sum of only 6 numbers, $a_0,\ldots,a_5$. This calculation could be performed with a single substitution statement, namely

$$s = a_0 + a_1 + a_2 + a_3 + a_4 + a_5$$

Clearly, this approach, although applicable to the more general problem, would require a different substitution statement for each value of n. A segment of flowchart illustrating this approach is shown in Figure 1.6, where 101 different substitution boxes would be required for this problem. There must be a better way!

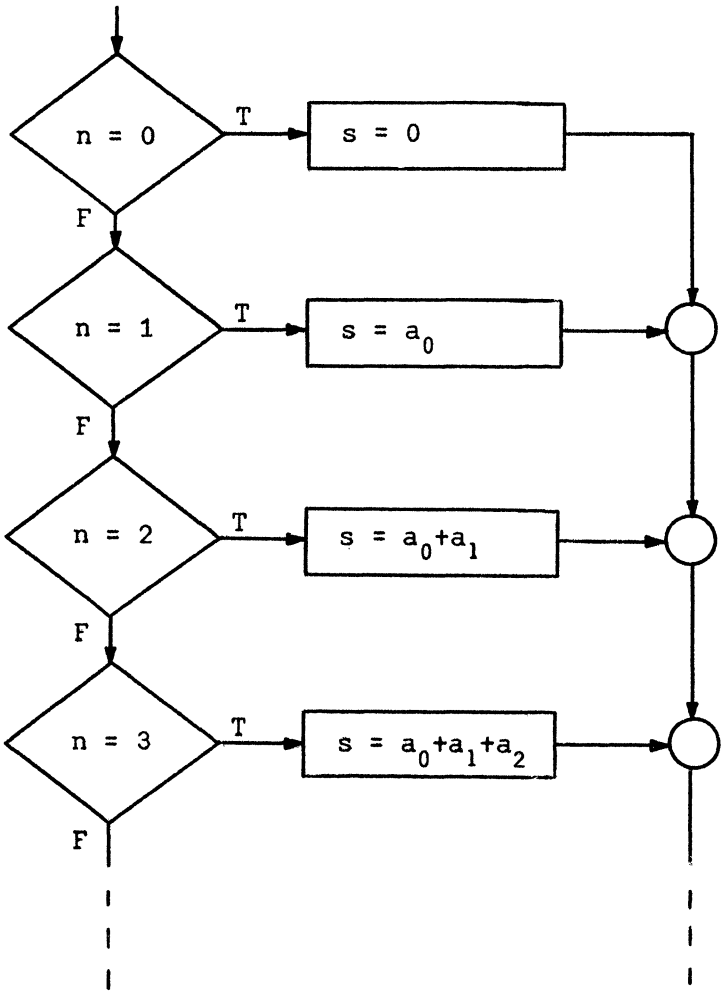



Figure 1.6

Let us consider another approach to the problem, namely that which one might use with an adding machine. For this method of solution, the total is first set to zero; then, the first number is added in; then the second number is added in; and so forth. The statements for this approach might be:

$$
\begin{aligned}
s &= 0 \\
s &= s + a_0 \\
s &= s + a_1 \\
s &= s + a_2 \\
s &= s + a_3 \\
s &= s + a_4 \\
s &= s + a_5
\end{aligned}
$$

Although this method requires many more statements in MAD, the machine language program for this method is quite similar to that for the single statement method. Next, we can carry this idea one step further by introducing the concept of an index, i, to keep track of which number is being added in. The statements become

$$
\begin{aligned}
s &= 0 \\
i &= 0 \\
s &= s + a_i \\
i &= i + 1 \\
s &= s + a_i \\
i &= i + 1 \\
s &= s + a_i \\
i &= i + 1 \\
s &= s + a_i \\
i &= i + 1 \\
s &= s + a_i \\
i &= i + 1 \\
s &= s + a_i \\
i &= i + 1
\end{aligned}
$$

where the need for the last statement becomes apparent below. Although the modifications have doubled the number of statements, each pair of statements (after the first pair) is the same. Instead of repeating the statements they can be included in a loop, where a decision is introduced to terminate the loop after executing

these statements the required 6 times. A flowchart for this process is shown in Figure 1.7. The execution of the statements in the loop is terminated when the

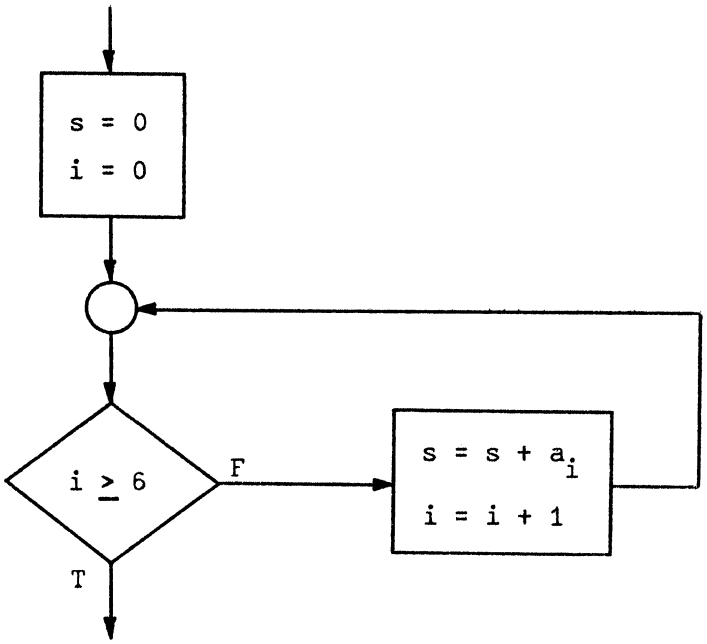



Figure 1.7

index reaches 6, i.e. just after $a_5$ is added in. A modification of this procedure so as to add n numbers requires merely changing the terminating condition to $i \geq n$. A MAD routine for the complete algorithm is shown in Figure 1.8a and the corresponding flowchart in Figure 1.8b.

```
          R   RØUTINE FØR THE SUM ØF N NUMBERS
           INTEGER N,I
           DIMENSIØN A(99)
START      READ AND PRINT DATA          N,A(0)...A(N-1)
           S = 0.0
           I = 0
LØØP       WHENEVER I .GE. N, TRANSFER TØ PRINT
           S = S + A(I)
           I = I + 1
           TRANSFER TØ LØØP
PRINT      PRINT RESULTS S
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 1.8a

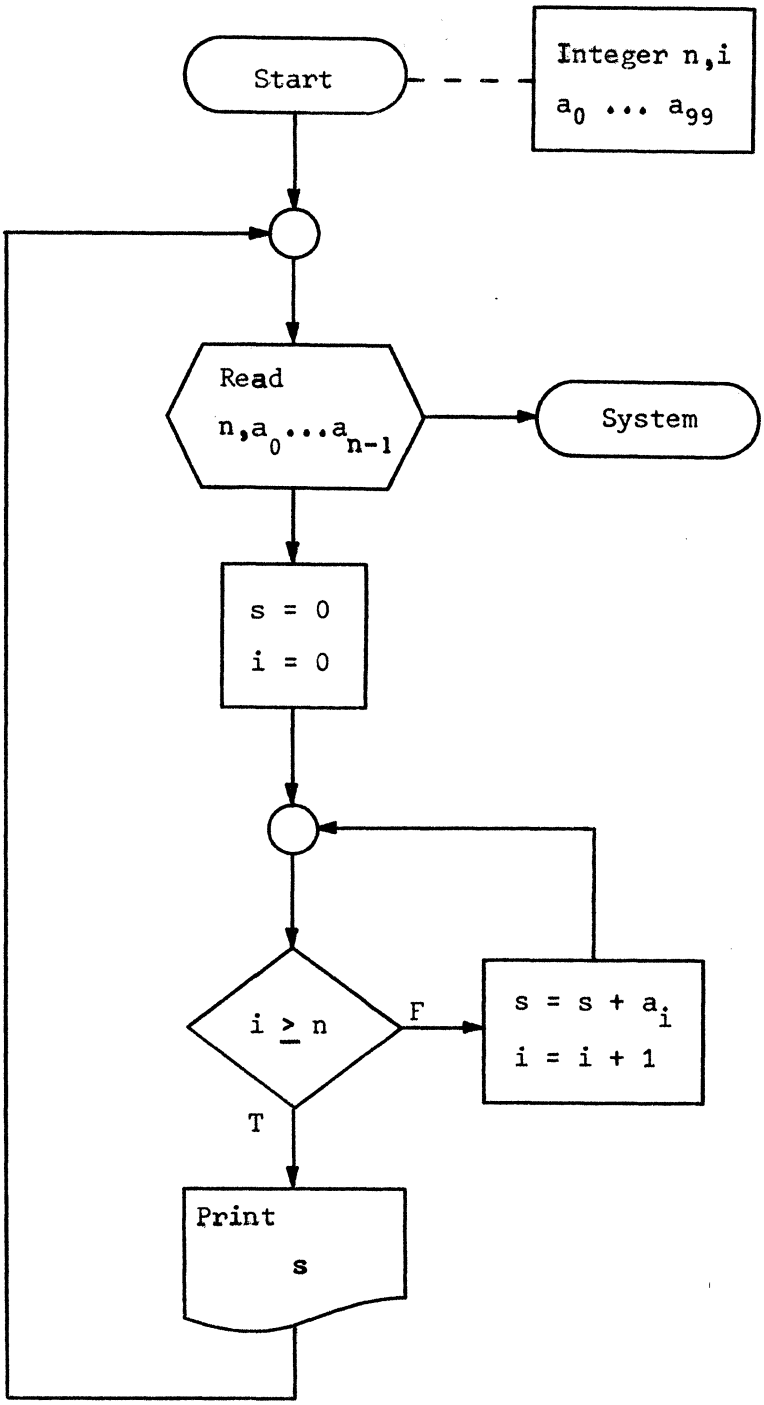

Start
Integer n,i
$a_0 \ldots a_{99}$
Read
$n, a_0 \ldots a_{n-1}$
System
s = 0
i = 0
$i \geq n$
F
$s = s + a_i$
$i = i + 1$
T
Print
s

Figure 1.8b

Note that this routine uses the INTEGER and DIMENSIØN declarations. They have been placed at the beginning of the routine in this example, although that choice is arbitrary.

The operation of this routine can best be illustrated by considering the sequence of events resulting from two typical sets of data, which follows:

```
N = 2, A(0) = -6.7, 2.3*                                (printed out)

                    N = 2
                    A(0) = -6.7
                    A(1) = 2.3
                    S = 0
                    I = 0
                    0 .GE. 2          (false)
                    S = -6.7
                    I = 1
                    1 .GE. 2          (false)
                    S = -4.4
                    I = 2
                    2 .GE. 2          (true)

                S = -4.400000                           (printed out)

N = 4, A(0) = 5.2, 7.3, 8.1, -2.9*                      (printed out)

                    N = 4
                    A(0) = 5.2
                    A(1) = 7.3
                    A(2) = 8.1
                    A(3) = -2.9
                    S = 0
                    I = 0
                    0 .GE. 4          (false)
                    S = 5.2
                    I = 1
                    1 .GE. 4          (false)
                    S = 12.5
                    I = 2
                    2 .GE. 4          (false)
                    S = 20.6
                    I = 3
                    3 .GE. 4          (false)
                    S = 17.7
                    I = 4
                    4 .GE. 4          (true)

                S = 17.700000                           (printed out)

****ALL INPUT DATA HAVE BEEN PROCESSED                  (printed out)
```

Only those lines that are indicated would actually be printed by the computer. In this example, only two sets of data were used and control was returned to the system after processing the second set. The last statement is printed by the system to indicate that an attempt has been made to read data, but none was found.

Finally, let us now examine this program to see that we have met the requirements for an algorithm. First, this algorithm is general; it can handle any size of n (with the only limit being the size of the block of memory reserved) and any set of real numbers. Second, it is conclusive; it will always terminate after a finite number of passes around the loop, namely n. Finally, it is definite; at each point in the algorithm, the next step is specified.

The structure of many loops in MAD routines can be greatly simplified by use of the iteration statement. The routine for the sum of n numbers is modified in Figure 1.9 to utilize that statement; the operation of the two routines is the same. A CØNTINUE statement is utilized in this version; it causes no action, serving merely as a labeling point. The THRØUGH statement initializes the value of the variable I to zero (the first value following the equal sign). The relation is next evaluated, and, if its value is <u>false</u>, all statements up to and including the one labeled END are executed. If the value of the relation is <u>true</u>, execution of the loop is terminated and the first statement <u>after</u> the one labeled END is executed next. Each time the loop is traversed and the labeled statement is executed, control is returned to the beginning of the loop, the variable is incremented (by the second value following the equal sign in the THRØUGH statement, 1 in this case), and the relation is evaluated. Thus, the loop is traversed until the value of the terminating

```
R     RØUTINE FØR THE SUM ØF N NUMBERS
      INTEGER N, I
      DIMENSIØN A(99)
START READ AND PRINT DATA      N, A(0)...A(N-1)
      S = 0.0
      THRØUGH END, FØR I = 0, 1, I.GE.N
          S = S + A(I)
END   CØNTINUE
      PRINT RESULTS S
      TRANSFER TØ START
      END ØF PRØGRAM
```

Figure 1.9

condition is true. The flowchart of Figure 1.8b is still valid (although a more convenient form will be introduced in Chapter 6) and the sequence of events is still essentially as shown for the first version of the routine.

## Exercises

The following set of exercises is suitable for testing some simple algorithms on the computer. By choosing one part of each problem, the reader will have a graded set of exercises. Note: If you have never programmed a digital computer, do not skip the first few exercises because they appear too easy. They are designed to teach you how to communicate with a computer without the need to worry about programming difficulties.

1. Draw a flowchart and write a MAD routine to solve each of the following problems:

   a) Given the base, B, and the height, H, of a triangle (where B and H are non-negative), calculate the area.

   b) Given three numbers, A, B, and C, calculate their average.

   c) Given the radius, R, of a circle, calculate the area.

In each case, assume all variables are in floating point mode and in each case, call the result of the calculation ANSWER. Your routine should include the following cards plus one for the calculation:

```
START       READ AND PRINT DATA
            PRINT RESULTS ANSWER
            TRANSFER TØ START
            END ØF PRØGRAM
```

Be sure that you use sufficient data to completely test your routine.

2. Repeat Exercise 1 for the following problems. Each routine involves a branch and a comment in addition to the steps of the first routine. All variables are in floating point mode.

   a) If the input, X, is positive, calculate and print out its square root. Otherwise, print a comment "X is negative. The square root is imaginary." and do not print any results.

   b) If the magnitude of the input, A, is less than 10., calculate and print out 5 + .5A. Otherwise, print only the comment "saturation".

   c) The inputs are CØST and PAID. If CØST is greater than PAID, print only the comment "You do not have enough money." Otherwise, calculate and print the change, PAID - CØST.

3. Repeat Exercise 1 for the following problems. In each problem, the input includes a floating point mode vector, A, consisting of three components A(0), A(1), and A(2). The output, to be called ANSWER, is in integer mode. This routine must include the appropriate declarations. The calculations or manipulations each take several steps. Do not use a loop.

a) Obtain the subscript of the algebraically largest component of A. If two or three A's are the same, ANSWER should be the smallest subscript from that set of two or three. (Be sure that this routine works when all A's are negative.)

b) The data also includes a floating point number Q. Print the subscript of the first A that is within .01 of Q (i.e. $|A(I) - Q| \leq .01$). If none of the A's meet that condition, print a comment to that effect. (Be sure that the program works when all or none of the A's are within .01 of Q.)

c) The data also includes a floating point number Q. Print the total number of A's (0, 1, 2, or 3) that are greater than or equal to Q. (Be sure to test the program for ANSWER = 0 and some A(I) equal to Q.)

4. Repeat Exercise 3, using a loop. Then extend the generality of the routine to handle a vector of N components (where N is in integer mode and less than 100).

5. Repeat Exercise 1 for the following problems where N is in integer mode, $0 < N \leq 100$, and the output, ANSWER, is in floating point mode.

a) Calculate and print out the product of N floating point mode numbers, A(1) ... A(N).

b) Calculate and print out N! (N factorial).

c) Calculate the dot product of two N dimensional floating point mode vectors, A(1) ... A(N) and B(1) ... B(N), i.e. $\sum_{i=1}^{n} a_i \cdot b_i$ .

## Chapter Two

## THE COMPUTING MACHINE

Thousands of electronic computers and data processors are now in use around the world and they influence almost every aspect of our lives. Although man has long recognized the need for such devices, these machines have been perfected only within the last thirty years. A computer is a machine which is capable of doing mental work, as opposed to the mainly physical work performed by earlier machines. In combination with physical type machines, a computer may serve as a sophisticated control mechanism to coordinate the operation and collect data on the functioning of the combined system. A computer may also be used as an adjunct to the human mind which extends the powers of thought and calculation. It is in this latter role that we will be concerned with the use of computers in this chapter.

Although the primary emphasis in this text is on the analysis of problems in order to develop algorithms for their solution, it is also intended that the descriptions of these algorithms should serve as instructions to a computing or data processing machine. Such a machine should then be able to apply these instructions to specific instances of the problem and produce the corresponding solutions or results. It is also true that some of these algorithms could be carried out by human beings; but in other cases these humans would not live long enough to complete some jobs which a machine can finish in a few minutes. Furthermore, many problems exist where the results are of no value unless they are available within some fairly short period of time. On the other hand, even the machines can be given jobs which would take them so long to accomplish, perhaps many years, that they would likely break down before the answers were obtained. Thus, although these machines may greatly extend our powers of computation and manipulation, they do not make these powers infinite.

Since the reader will probably be using a computer to test algorithms and perhaps later to produce solutions of important problems, it is helpful for him to know something about these machines before he proceeds to the later chapters. The use of such information should make it possible for him to write more efficient algorithms and to better judge what can be accomplished with the aid of these machines. In fact, the structure and organization of the machines often influences the way in which one goes about solving a problem.

In this chapter we shall first discuss the background and general organization of computers. Next, we present some details on data representation in machines and on the various components of the equipment. Finally, we discuss the ways of using a machine including how it may be programmed in various types of language.

## 2.1 Background

Throughout history intelligent men have been striving to extend their powers. Tools, weapons, beasts of burden, slaves, and harnessed forces of nature were early results of this activity. It has long been recognized that many of the functions, operations and other actions, both physical and mental, which are carried out by men are largely mechanical in nature. At the same time man has realized his own propensity to make errors in even those operations which he completely understands how to carry out. Such considerations have driven man to develop mechanical aids to help in his work. In many cases, these devices have completely eliminated some of the processes that man previously performed and provide capacities to do work which greatly surpass the native abilities of man.

Our major interest concerns the development of processes involving computation with numbers and manipulation of other symbols. There are, however, many historic developments which have contributed to developments in this area, although not directly related to it. We will therefore trace briefly the historical development of computation, but at the same time mention some of these contributing factors.

The use of notches, stones, marks, symbols, pictures and other aids to memory had to be developed before history could even be recorded. Using such symbols to record numbers was fairly natural, but the development of the Arabic notation was certainly a major achievement. This Arabic representation made possible the invention of the abacus which has provided a mechanical aid to computation so valuable that its use persists today. Much later, in 1642, the French scientist Pascal invented an adding machine with numbered notches along the edges of wheels, and pins that caused automatic carries from one position to the next. This idea was extended in 1671 by the German mathematician Leibnitz who designed a machine that could also do multiplication. He wrote at that time, "It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could safely be relegated to anyone else if machines were used."

Clocks -- another aid to human thought and activities -- were one type of machine which progressed quite ahead of most others, and it became quite popular to add intricate embellishments to their mechanisms. Some of these novelties on clocks involved figures of people or animals, so that it was natural to extend

this idea and develop devices exclusively to control the behavior of robots. The Parisian automaton maker Vaucanson became famous for such work in the 18th Century. Although devices of this kind served mainly to entertain, as they do even today, the techniques developed did contribute and lead to the perfection of useful automata.

Another area in which machines progressed fairly early was in the weaving trade. Joseph Jacquard, one of a family of French silk weavers, applied some of Vaucanson's techniques and developed a new type of loom which was first exhibited in Paris in 1801. This loom used cards with a pattern of holes punched in them to control the design which was woven, but it needed an artificial source of power for its successful development. The steam engine perfected by Newcomen and Watt in England furnished such power and served to initiate the industrial revolution.

Calculating machines such as the abacus, the machines of Pascal and Leibnitz, and, for that matter, the more modern electric adding machines and desk calculators had a very basic limitation in their function. They could be used to carry out only relatively simple operations and required a person to operate them. This person not only had to enter the numbers as they were needed and activate the various steps in the process, but also had to record intermediate results frequently and organize the final answers into tables, charts or graphs. The possible speed of operation, and the accuracy or reliability of such a device was therefore limited severely by the person who operated it.

In order that a machine be able to independently carry out a complex and varied sequence of operations, it was necessary for it to be able to accept long sets of detailed instructions, store intermediate results, convert its instructions into the individual operations and eventually produce the desired results. Realizing all these things, an English mathematician named Charles Babbage devoted much of his life to designing and attempting to build computing machines. His first machine, called the Difference Engine, was completed in 1822 and was designed to evaluate polynomials. His effort to build a larger machine of that type failed because of the inadequate machining techniques of that day. Undaunted, Babbage extended his ideas and by 1833 had developed plans for a new machine, called the Analytical Engine. He worked on the perfection and construction of this device for the rest of his life, almost forty years, but was unable to complete even a substantial part of it. This machine would have represented input data and instructions using the idea of punched cards from the Jacquard loom. The engine would have been able to store a thousand fifty-digit decimal numbers. Instructions would be carried out sequentially and it would have been able to go backward or forward in this sequence based

on the results. As we shall see, Babbage conceived most of the features found in a modern computer, although he did not have available to him the technology to convert these ideas into operating systems.

The effective development of punched cards for representing data was the work of Dr. Herman Hollerith in the 1880's. His machines were used to compile the U. S. Census of 1890 and 1900, and were later commercially developed by what is now the International Business Machines Corporation. Similar machines were developed by James Powers in 1907 for the census of 1910 and lead to the development of part of what is now the Sperry Rand Corporation. Since Hollerith's punched-card machines used electricity to sense the presence of a hole in a card and later to drive the mechanism which moved cards from one position to the next, they would not have been possible in Babbage's day.

Additional machines for punching, sorting and collating cards as well as tabulating data from them were developed over the years. In time, the tabulators could add and multiply numbers from the cards as well as store a few numbers. These machines were programmed by plugging wires into plug-boards which were then inserted into the machine to control its function. Building on these techniques of reading holes, relays, plug-boards, contact wheels and mechanical devices, Professor Howard Aiken of Harvard University in cooperation with IBM began construction of the Automatic Sequence Controlled Calculator, Mark I, in 1939. Dr. Aiken used many of Babbage's ideas in his machine, but had discovered a great number of them independently before he came across the previous work. The Mark I was completed in 1944 and used to compute many important tables. During the same period, several other relay type calculators were under development, but better methods were soon to replace them.

The foundations for one of these methods were laid by an 18th Century English mathematician, George Boole, whose work in symbolic logic resulted in the development of an algebra which bears his name. A special case of this algebra serves as the basis for the logic of modern computers. An English logician, A. M. Turing, also made significant contributions to the mathematical theory of machines starting in the 1930's.

The development of electronics was also necessary to the implementation of the new methods. In 1919, Eccles and Jordan, developed an electronic trigger circuit which could maintain either one of two stable states. This device, called a bistable multivibrator or flip-flop, is able to store information. The area of electronics began to develop rapidly before and under the pressure of World War II and it was discovered that the algebra of Boole could be implemented using electronic

tubes in circuits which would function much faster than electro-mechanical relays. At the Moore School of Electrical Engineering of the University of Pennsylvania it was decided to build a machine similar to Mark I, but using electronic circuits instead of electro-mechanical methods. This machine, called the ENIAC for Electronic Numerical Integrator and Calculator, was designed by Professors J. Eckert and J. Mauchly. ENIAC, like its predecessors, was very difficult to set up and to program; it usually took several hours to rewire it for a new job. Completed in 1946, ENIAC was too late to help with the war effort which had stimulated its creation, but was used for numerous important calculations.

During the construction of ENIAC, three mathematicians at the Institute for Advanced Study, Major Herman Goldstine, Professor Arthur Burks, and Professor John von Neumann, conducted a careful study of the logic of computing machines and published a number of reports. These reports proposed many new ideas including the use of binary representation for numbers to improve the efficiency of the machines, but the most important idea of all was that of a stored program. This idea, credited to von Neumann, involves the storing of a program of instructions in a common memory with numbers and other data. This technique enabled the computer to operate on the program in much the same way that previous machines had operated on numbers. Such stored program computers could write or modify their own instructions and also have their programs changed very easily. No other development has contributed more to the success of computers than the idea of the stored program.

Numerous machines using these new techniques were soon under construction, including the SEAC at the National Bureau of Standards, the EDVAC at the Moore School, the I.A.S. machine at Princeton, and the EDSAC at the University of Cambridge, England, among others. Eckert and Mauchly had left the University of Pennsylvania to form their own company, and produced the first commerical machine called UNIVAC in 1951. Engineering Research Associates soon marketed a machine called the ERA 1101. Both these firms later became part of Sperry Rand. IBM followed shortly with several types of machines and was soon the strong leader of the industry.

Technological advances in components, fabrication, design, programming and other factors have occurred so rapidly that machines were usually outdated before they could be built. In fact, the cost of computing power decreased so rapidly that it was often not economical to accept an old computer, even as a gift. A significant advance was the development of solid state devices such as transistors, diodes and memory cores to replace vacuum tubes and other earlier components. These new components have brought about increased reliability, reduced size, higher speeds, and lower costs. Other advances will be discussed in later sections.

There are now over 30,000 general purpose computers in use in the United States alone. This figure does not include the thousands of special purpose machines used in military applications, in the space program, or in industrial control. Numerous other manufacturers have joined IBM and Sperry Rand in competition for a still rapidly expanding market. In the United States these manufacturers include Control Data Corporation, General Electric, Radio Corporation of America, Burroughs, Honeywell, Scientific Data Systems, Digital Equipment Corporation, National Cash Register, Bunker-Ramo Corporation, Philco, Computer Control Company, Advanced Scientific Instruments, and several smaller firms. Numerous other organizations in England, France, Germany, Russia, Sweden, Belgium, Netherlands, Czechoslovakia, Poland, Hungary, Yugoslavia, Italy, and Japan also are active in the manufacture of computers.

Since the range of computer applications broadens daily and the work load for most uses which are already established is still growing rapidly, it is not yet possible to foresee any saturation point for progress in the development of computers and their applications. Almost all of the predictions which have been made on this subject during the last decade have been woefully shortsighted.

## 2.2 Organization

An automaton in the modern technical sense is a device which can accept inputs, process them and produce outputs which are dependent on its present and past inputs. Many machines, including the lowly thermostat and the traffic signal lights at the corner, may thus be classified as automata. Computing machines are automata, but obviously not all automata are computing machines. In particular, a computer is a versatile automaton designed to handle, manipulate and perform calculations on data consisting of numbers or symbols. In an analog computer numbers are represented by essentially continuous physical entities or quantities such as a voltage or the angular position of a wheel. Thus, a slide rule is an elementary analog computer. In the computers considered in this text, however, numbers and symbols are represented by discrete entities like digits, switches, beads or notches. These machines are called digital computers or data processors. An abacus is therefore a simple digital calculating device. Today, we often find digital and analog devices combined into a single system which is then called a hybrid computer.

There are many specially designed digital and analog devices which are used in various applications to control mechanical equipment, but which are not necessarily computers. However, there is a tendency today to use a more flexible digital computer for such control applications, thus making it part of a larger automaton which is mainly devoted to performing physical type work. For example, a computer

may be used to keep a missile on course, to land an airplane in fog, to run a chemical plant, to control machine tools in an automobile factory, or to handle the switching of telephone calls. The central control of such systems consists of a digital computer with analog devices to interface with the physical equipment. On the other hand, a computer may be relatively free of such connections and used primarily by people to extend their powers of computation and manipulation. It is this latter general purpose use of computers to which our discussion will henceforth be devoted. It is possible, of course, for the same computer to be shared among a wide variety of activities of several different types.

We have seen that there are several basic abilities which a machine--or for that matter any other organization--must have in order that it be able to carry out a variety of processes which may not be specified until long after the machine is constructed. First, it must be able to communicate with the outside world. That is, it must be able to receive as input both the raw materials of these processes and the instructions as to how the processes are to be done. Second, it must be able to perform certain basic operations and to combine these into new more complex operations according to instructions given to it. Third, it must be able to convey the results of these processes to the other people or machines with which it works.

In the following discussion of the general organization of computer systems, many terms and ideas will be used with little or no explanation, but later sections will contain more information about these concepts. In this material, we will discuss features of computers in general, indicating the wide variation in complexity and philosophy which is found in today's machines, rather than limiting ourselves to any particular existing machine.

A modern general purpose electronic digital computer system consists of several basic parts. It has input devices which can read or accept properly prepared material such as punched cards, punched paper tape, magnetic tape, and perhaps even printed pages, verbal commands, or signals from various other devices. It has output devices which can, for example, write information on magnetic tape, print it on paper, punch it in cards or paper tape, draw graphs or pictures, produce photographs or now even give verbal responses in a natural language such as English. It can often communicate directly with people through remote stations consisting of automatic typewriters or other such devices. It may even communicate with other machines. It has a large memory or storage in which representations of numbers, letters, instructions, words and other symbols may be kept. It has at least one processor with the ability to follow instructions and perform a variety of operations, such as storing in memory, fetching from memory, addition, subtraction,

multiplication, division, comparison, controlling input-output devices, and many others. Finally, it has control mechanisms which coordinate the functioning of its various parts. The control function is performed mainly in the processors which are capable of sequentially carrying out instructions that have been stored in the memory in a coded form. Thus, a processor can carry out a lengthy sequence of operations according to some preconceived plan. In fact, it can make decisions based on the progress of its own work and thus take alternate paths through its instructions.

In most early computers and some smaller computers today, a single processor serves as the center of the system with all input-output passing through it on the way to and from memory. In such systems, a single control sequencer is used for both internal processing and input-output. With this arrangement, depicted in Figure 2.1, the machine is either performing internal operations or communicating with the input-output devices but never both at the same time. (In the machine organization diagrams of this chapter solid lines will be used to indicate data flow and dashed lines to indicate control signals between the units.)

Since the input-output operations are usually relatively slow, compared with internal operations, considerable economy may be achieved by separating these activities as much as possible. Thus, most systems are now centered around the memory and the control of internal processing is partially separated from that of input-output processing. Figure 2.2 illustrates an organization in which the processor and the input-output units communicate independently with the memory under a common control.

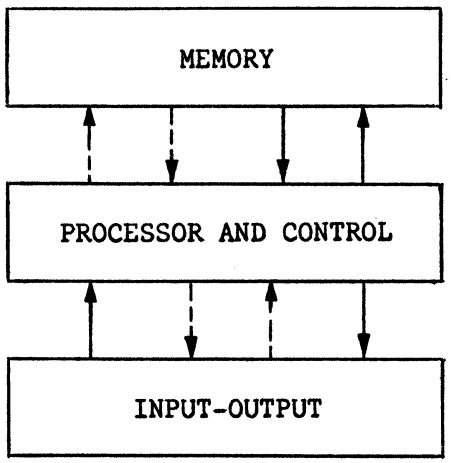



Figure 2.1 A simple organization

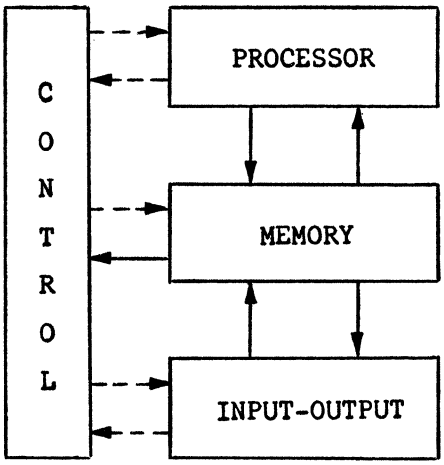



Figure 2.2 A better organization

In larger systems a central or main processor is used almost exclusively for internal operations on data contained in a central or main memory and can proceed on these operations at full speed, largely independent of the input-output functions. Special processors, often called input-output controllers (I-O controllers), are then used to control the input-output operations and also to transfer data between main memory and slower storage units. In many cases, smaller satellite computers are used to do the large volumes of basic input-output and data is transferred between these computers and the larger computers by switching magnetic tapes or disks. Figure 2.8 shows such an organization with one I-O controller and one satellite

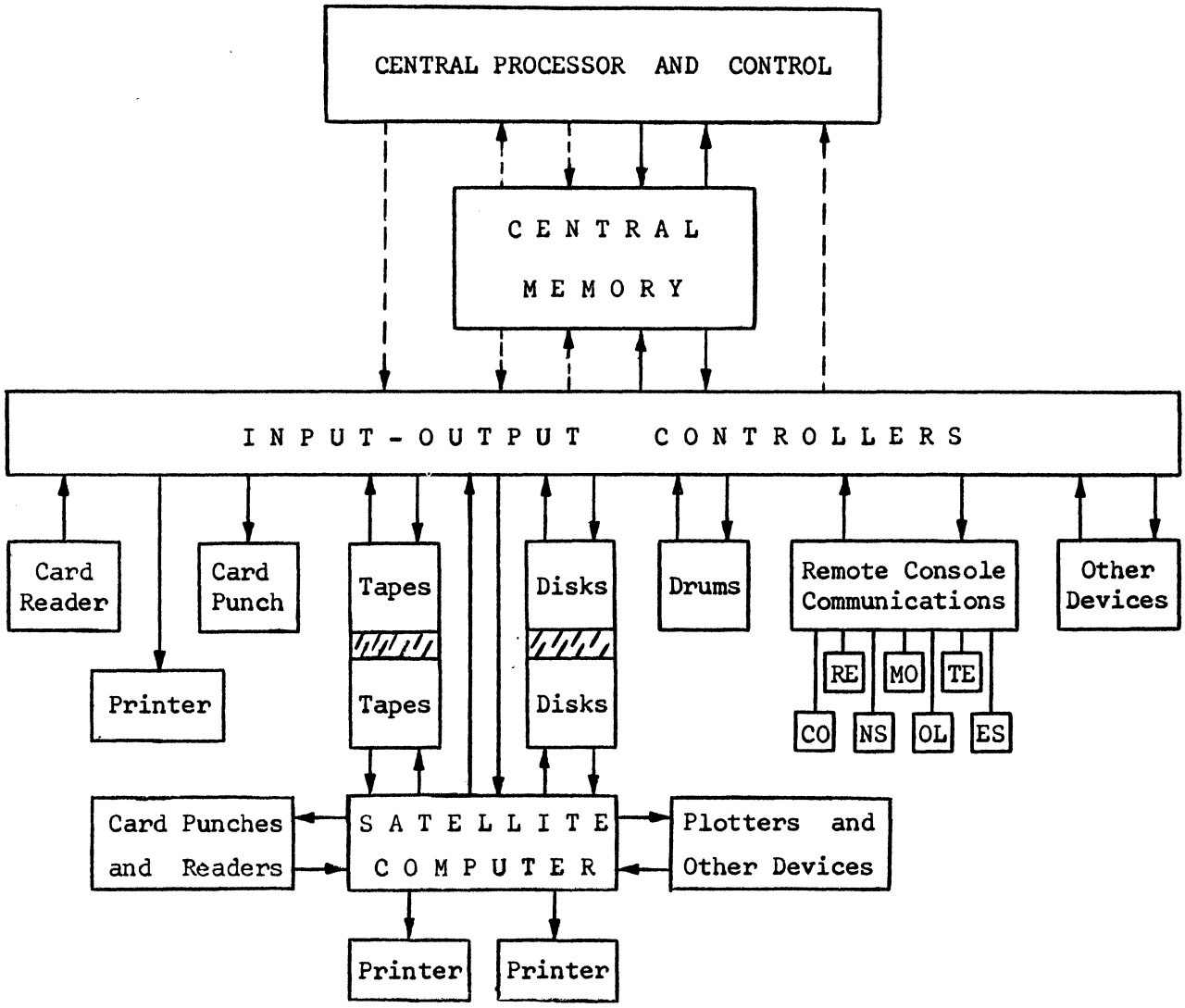



Figure 2.3 A typical large computer organization

computer. Often more than one of these satellites are used and sometimes their functions are replaced by one or more computers directly coupled to the main memory.

Some systems of recent design allow several main processors to operate out of a common central memory. Other systems also use small ancillary computers operating out of this memory to communicate directly with the input-output devices and transfer data between main memory and various auxiliary and subsidiary storage units. The main problem in developing and using such a system is the coordination of the various activities so that they do not interfere with one another. Often this problem is solved by designating a particular processor, not always the most powerful one in the system, as the coordinator or master of the system. It is also possible to incorporate special purpose processors into such a system.

## 2.3 Binary Codes

The numbers, letters, characters, instructions and other data to be used by a computer must be represented in a form which the computer can read and manipulate. For this purpose, almost all computers use devices which can have exactly two states, usually designated by 0 and 1. For example, computers frequently read in information from cards or paper tape into which holes may be punched at certain locations (representing a 1), or not punched (representing a 0). Internally, computers frequently use the two states of a flip-flop to represent either 0 or 1. Numerous examples of other two-state devices will be seen in the later sections of this chapter, but the immediate objective is to show how various symbols can be represented in terms of zeros and ones. Since 0 and 1 are the digits of the binary number system, symbols represented in this manner are frequently said to be binary-coded. The term binary digit is usually shortened to bit. A review of binary and other number systems is given in Appendix B.

By using n binary or bi-stable devices it is possible to represent a maximum of $2^n$ different symbols or states. Thus, to represent the ten decimal digits, at least four such binary choices must be used. The correspondence between the symbols and their binary coding may take many forms. Table 2.1 gives a few of the many possible codings which have been used for the binary coding of decimal digits using exactly four bits. The digital designations such as 8421 indicate the value of the corresponding bits in the code. Note that in both the 7421 and 6321 codes more than two ones in a representation could be used to indicate an error. In the Excess 3 code, the codings for two digits whose sum is 9 simply have their zeros and ones interchanged. This latter property was used to simplify the arithmetic in some early machines.

Table 2.1 Four bit codes for decimal digits

| Decimal Digit | Octal | Binary | Binary-Coded Decimal | | | |
|---|---|---|---|---|---|---|
| | | | 8421 | 7421 | 6321 | Excess 3 |
| 0 | 0 | 0 | 0000 | 0000 | 0000 | 0011 |
| 1 | 1 | 1 | 0001 | 0001 | 0001 | 0100 |
| 2 | 2 | 10 | 0010 | 0010 | 0010 | 0101 |
| 3 | 3 | 11 | 0011 | 0011 | 0100 | 0110 |
| 4 | 4 | 100 | 0100 | 0100 | 0101 | 0111 |
| 5 | 5 | 101 | 0101 | 0101 | 0110 | 1000 |
| 6 | 6 | 110 | 0110 | 0110 | 1000 | 1001 |
| 7 | 7 | 111 | 0111 | 1000 | 1001 | 1010 |
| 8 | 10 | 1000 | 1000 | 1001 | 1010 | 1011 |
| 9 | 11 | 1001 | 1001 | 1010 | 1100 | 1100 |

In each of the four bit codes of Table 2.1 only ten of the sixteen possible codings are used and so these codings contain a certain amount of redundancy. This redundancy could be used by computer designers to enable the machine to signal that an error has occurred when one of the six illegal codings is detected. If more than four bits are used, more complete error detection can be obtained as illustrated by the five codes listed in Table 2.2. Parity codes use a fifth bit which is chosen to make the number of ones in the code either even or odd. The two-out-of-five codes make use of the fact that there are exactly ten combinations of five things taken two at a time, and thus the code for each digit contains exactly two ones. The bi-quinary code uses a total of seven bits to represent a single decimal digit. In each of these cases, any error causing a change of a single bit, as well as many other errors, can be detected. All of these codes (and even more complex schemes) have been used in machines.

In order to represent letters and other symbols in addition to decimal digits, it is necessary to use at least six bits in the code. Actually, only 48 of the 64 symbols which can be represented by 6 bits have usually been used to represent the

Table 2.2 Redundant binary codes for decimal digits

| Decimal Digit | Parity Codes | | 2-out-of-5 Codes | | Bi-Quinary Code |
|---|---|---|---|---|---|
| | Even P8421 | Odd P8421 | 74210 | 63210 | 50 43210 |
| 0 | 00000* | 10000 | 11000† | 00110† | 01 00001 |
| 1 | 10001 | 00001 | 00011 | 00011 | 01 00010 |
| 2 | 10010 | 00010 | 00101 | 00101 | 01 00100 |
| 3 | 00011 | 10011 | 00110 | 01001 | 01 01000 |
| 4 | 10100 | 00100 | 01001 | 01010 | 01 10000 |
| 5 | 00101 | 10101 | 01010 | 01100 | 10 00001 |
| 6 | 00110 | 10110 | 01100 | 10001 | 10 00010 |
| 7 | 10111 | 00111 | 10001 | 10010 | 10 00100 |
| 8 | 11000 | 01000 | 10010 | 10100 | 10 01000 |
| 9 | 01001 | 11001 | 10100 | 11000 | 10 10000 |

*Sometimes 01010 is used in order that the bits will not all be zero.
†The values do not add up to the digit in these cases.

26 upper case letters, 10 digits, 11 special characters and the blank. Some common codings for these 48 characters are given in Table 2.3. Since there is a simple correspondence between one octal digit and three binary digits (see Appendix B), it is common to write the octal equivalent of binary codes. The storage and BCD tape codes of Table 2.3 are represented in this way. The storage code listed here is the one used in the memory of the IBM 7040 and 7090. The BCD tape code is a standard code used by the industry for recording data on magnetic tape. In the memory code, the alphabetic order of the letters agrees with the numeric order of their codes, thus making the sorting of information into alphabetic order more efficient on the machine. The card code will be discussed further in the section on input and output. The odd parity BCD code is a common code for paper tape and is the internal storage code of the IBM 1401. In this column the binary ones are indicated by having the name of the corresponding column printed. The parity or check bit is

Table 2.3 Some Codes for 48 Common Characters

| Character | Octal Codes | | Card | Odd Parity |
|---|---|---|---|---|
| | Storage | BCD Tape | Code | BCD Code |
| 0 | 00 | 12 | 0 | C 8 2 |
| 1 | 01 | 01 | 1 | 1 |
| 2 | 02 | 02 | 2 | 2 |
| 3 | 03 | 03 | 3 | C 21 |
| 4 | 04 | 04 | 4 | 4 |
| 5 | 05 | 05 | 5 | C 4 1 |
| 6 | 06 | 06 | 6 | C 42 |
| 7 | 07 | 07 | 7 | 421 |
| 8 | 10 | 10 | 8 | 8 |
| 9 | 11 | 11 | 9 | C 8 1 |
| = | 13 | 13 | 38 | 8 21 |
| ' | 14 | 14 | 48 | C 84 |
| + | 20 | 60 | + | CBA |
| A | 21 | 61 | +1 | BA 1 |
| B | 22 | 62 | +2 | BA 2 |
| C | 23 | 63 | +3 | CBA 21 |
| D | 24 | 64 | +4 | BA 4 |
| E | 25 | 65 | +5 | CBA 4 1 |
| F | 26 | 66 | +6 | CBA 42 |
| G | 27 | 67 | +7 | BA 421 |
| H | 30 | 70 | +8 | BA8 |
| I | 31 | 71 | +9 | CBA8 1 |
| . | 33 | 73 | +38 | BA8 21 |
| ) | 34 | 74 | +48 | CBA84 |
| - | 40 | 40 | - | B |
| J | 41 | 41 | -1 | CB 1 |
| K | 42 | 42 | -2 | CB 2 |
| L | 43 | 43 | -3 | B 21 |
| M | 44 | 44 | -4 | CB 4 |
| N | 45 | 45 | -5 | B 4 1 |
| Ø | 46 | 46 | -6 | B 42 |
| P | 47 | 47 | -7 | CB 421 |
| Q | 50 | 50 | -8 | CB 8 |
| R | 51 | 51 | -9 | B 8 1 |
| $ | 53 | 53 | -38 | CB 8 21 |
| * | 54 | 54 | -48 | B 84 |
| (blank) | 60 | 20 | (blank) | C |
| / | 61 | 21 | 01 | C A 1 |
| S | 62 | 22 | 02 | C A 2 |
| T | 63 | 23 | 03 | A 21 |
| U | 64 | 24 | 04 | C A 4 |
| V | 65 | 25 | 05 | A 4 1 |
| W | 66 | 26 | 06 | A 42 |
| X | 67 | 27 | 07 | C A 421 |
| Y | 70 | 30 | 08 | C A8 |
| Z | 71 | 31 | 09 | A8 1 |
| , | 73 | 33 | 038 | C A8 21 |
| ( | 74 | 34 | 048 | A84 |

indicated by a C and the other bits by B, A, 8, 4, 2 and 1 respectively.

Usually, several different codings are used for characters within each computer. Often the code for one medium of storage or input-output device is not convenient or standard for another. Various historic circumstances have also caused a variety of codes to appear on various equipment. Although great efforts have been put forth to standardize codes, there are still many different ones being used. The reader should study Table 2.3 to learn the relationship between the various codes.

Although it has been customary to use only about 48 characters in computers, the tendency today is toward larger character sets so that 59 characters or more are now becoming available. With still more expansion in mind, the computer industry is gradually establishing eight bits as the standard for character representation which allows the representation of lower case letters and many other symbols.

In the interest of efficiency, it is usually desirable for computers to handle several bits at a time. Although systems have been designed which operate on as few as four bits at a time, most computers operate on units of eight or more bits and the use of as many as sixty-four bits at once is not uncommon. For purposes of classification we will call any machine which uses 8 or less bits at a time a character-oriented machine and most others word-oriented machines. We will call machines which have varied capacities mixed machines.

Exercises

1. Convert the following sets of octal words to character messages by dividing the 12 octal digits of each word into 6 pairs and looking them up in the storage column of Table 2.3:

| a) | b) | c) |
|---|---|---|
| 545454542143 | 214327465131 | 632122432560 |
| 436031454764 | 633044312360 | 023303606060 |
| 636024216321 | 442563304624 | 624644256023 |
| 603021652560 | 626064623145 | 462425626026 |
| 222525456047 | 276063302560 | 465160234644 |
| 514623256262 | 442124604321 | 444644602330 |
| 252433606060 | 452764212725 | 215121236325 |
| | | 516260606060 |

2. Convert the following character messages to the octal storage code of Table 2.3 writing the octal digits in words of 12 digits each:

   a) B + (C*X)/D + 3.4*W IS AN ARITHMETIC EXPRESSIØN.

   b) T = 4.71*SIN.(X).P.2 IS A SUBSTITUTIØN STATEMENT.

   c) TRANSFER TØ MARYLAND IS NØT A MAD STATEMENT.

## 2.4 Number Representation and Computer Arithmetic

There are several ways of representing numbers in a computer. First, the digits of a decimal number may be coded individually as shown in the previous section. For example, the number 183 might be coded as 0001 1000 0011. Alternatively, the number might be converted to binary representation (see Appendix B) and directly coded in binary. The number $183_{10}$ would then be represented by 10110111. Since at least 12 bits are necessary to separately code the digits of decimal numbers up to 999 and only 10 bits can represent binary numbers up to $2^{10} - 1$ or 1023, it is clear that the latter representation makes more efficient use of the bits. In fact, it takes only about 3.3 bits to represent a decimal digit of information converted to binary, as opposed to at least 4 bits for a BCD (binary coded decimal) representation. Although more efficient for internal representation and operations, the binary representation requires that a rather complex conversion be made on input and again on output so that the people working with the computer may use the familiar decimal representation. This conversion takes computer time, and if the internal processing is of a simple nature, it may be more economical to use a machine which encodes the digits individually. A machine which uses individually coded decimal digits when representing numbers and carrying out arithmetic operations is called a decimal machine, whereas a machine which does its arithmetic with numbers represented in strict binary form is called a binary machine. There are some machines with both binary and decimal capacities. The commonly used character-oriented computers are decimal machines, but both binary and decimal word-oriented machines are in use. Table 2.4 shows some of the characteristics of a number of machines. Although these are typical characteristics, in many cases they vary widely for a particular model of the machine. In addition, each manufacturer has other machines and there are other manufacturers.

Most scientific computations done on computers use a single fixed length word to represent each number. These fixed length words typically contain 7 to 20 decimal digits or 24 to 64 binary digits with digits not needed to represent a particular number being filled with zeros. This section will be mainly devoted to fixed word length representation of numbers and the operations on them. To be specific, consider a machine having 10 decimal digits, where, for example, some typical words might be:

2964380175 0000001492 0000000003

If a decimal point is assumed at the right hand end of such numbers all non-negative integers up to $10^{10}$ can be represented, but all computations must stay within these

Table 2.4 Some Machine Characteristics

| Manufacturer and Model | First Year | Digits /word $\alpha$ | Memory $\beta$ Size | Memory $\beta$ Cycle | Add $\zeta$ Time | Addresses $\eta$ | No. of Mach. $\kappa$ | Monthly Rental $\lambda$ |
|---|---|---|---|---|---|---|---|---|
| Burroughs Corp. | | | | | | | | |
| 205 $\tau$ | 1954 | W10d | $80+4K^{\gamma}$ | 850 | 1700 | 1 | 40 | 3-10 |
| B260 | 1962 | C1a | 9.6 | 10 | 777 | 3 | 230+ | 3-8 |
| B5500 | 1963 | W48b | 4-32 | 6 | 3- | 0-1 | 60 | 16-164 |
| Control Data Corp. | | | | | | | | |
| 160 | 1961 | W12b | 8-32 | 6.4 | 12.8 | 1 | 450 | 2-10 |
| 1604A | 1960 | W48b | 8-32 | 6.4 | 4.8- | 1 | 50+ | 30-50 |
| 3600 | 1963 | W48b | 32-262 | 1.5 | 2- | 1 | 50+ | 40-65 |
| 6600 | 1964 | W60b | 16-262 | .7 | .7 | 1 | 17+ | 60-140 |
| Digital Equipment | | | | | | | | |
| PDP-6 | 1964 | W36b | 16-262 | .5-4 | 3.5 | 1 | 23+ | 6-30 |
| PDP-8 | 1965 | W12b | 4-32 | 1.5 | 3.0 | 1 | 500+ | .5-3 |
| General Electric | | | | | | | | |
| 210 | 1959 | W6d | 4-8 | 32 | 64 | 1 | 50 | 10-35 |
| 235 | 1961 | W12d | 4-16 | 6.5 | 78 | 1 | 70+ | 6-26 |
| 435 | 1964 | W24b | 32 | 2.7 | 8.8 | 1 | 40+ | 8-25 |
| 635 | 1964 | W36b | 32-262 | 1- | 1.8- | 1 | 35+ | 40-100 |
| Honeywell | | | | | | | | |
| H-120 | 1966 | W12d-48b | 2-32 | 3 | 66 | 3 | 340+ | 2-4 |
| H-200 | 1964 | W12d-48b | 4-65 | 2 | 44 | 3 | 720+ | 3-10 |
| H-2200 | 1966 | W12d-48b | 16-260 | 1 | 22 | 3 | 50+ | 5-30 |
| IBM Corporation | | | | | | | | |
| 650 $\tau$ | 1954 | W10d | $60+4K^{\delta}$ | 100+ | 700 | 1 | 200+ | 3-10 |
| 701 $\tau$ | 1953 | W18-36b | 4 | 12 | 36/60 | 1 | 1 | 5 |
| 704 $\tau$ | 1955 | W36b | 4-32 | 12 | 24 | 1 | 30 | 25-50 |
| 1401 | 1960 | C1a | 1.4-16 | 11.5 | 230 | 2 | 8000+ | 2-12 |
| 1620 | 1960 | C1d | 20-60 | 10-20 | 140+ | 2 | 1700+ | 1.6-5 |
| 7070 | 1960 | W10d | 5-10 | 6 | 60- | 1 | 340+ | 12-31 |
| 7090 | 1959 | W36b | 32 | 2.18 | 4.36 | 1 | 45 | 60-100 |
| 7094 I&II | 62/4 | W36b | 32 | 2/1.4 | 4/2.8 | 1 | 260+ | 66-131 |
| Stretch | 1961 | W64b | 16-262 | 2.2 | 1.5 | 1 | 7 | 150-200 |
| 360/30 | 1965 | 1a | 8-64 | 2 | 39 | 0,1,2 | 4300+ | 3-20 |
| 360/50 | 1965 | M 32b | 64-256 | 2 | 4 | 0,1,2 | 400+ | 14-55 |
| 360/75 | 1965 | 64b | 256+ | 1- | 2 | 0,1,2 | 50+ | 45-170 |
| Philco | | | | | | | | |
| 2000-210 | 1958 | W48b | 8-32 | 10 | 15- | 1 | 19 | 20-60 |
| RCA | | | | | | | | |
| 301 | 1961 | C1a | 10-40 | 7 | 98 | 2 | 650+ | 3-25 |
| 501 | 1959 | C1a | 16-262 | 15 | 360- | 2 | 100+ | 11-26 |
| Spectra 70 | 1965 | C1a | 16-524 | .84-2 | 10-26 | 0,1,2 | 300+ | 4-60 |
| SDS Inc. | | | | | | | | |
| 910/920 | 1962 | W24b | 2-16 | 8 | 16 | 1 | 300+ | 1.5-6 |
| Sigma 7 | 1966 | M32b-1a | 4-131 | .7-1.2 | 2 | 1 | 14+ | 6-30 |
| Univac | | | | | | | | |
| I & II $\tau$ | 51/7 | W6-12d | $1/10^{\epsilon}$ | 200/20 | 525/160 | 1 | 20 | 20-30 |
| SS80 | 1958 | W10d | 1.2 | 17 | 51 | 1 | 250 | 4-13 |
| 1004 | 1963 | C1a | .961 | 6.5 | 91 | 1 | 3350+ | 1-3 |
| 1107 | 1962 | W36b | 32-65 | .6/4 | 4- | 1 | 32 | 32-80 |
| 1108 | 1965 | W36b | 65-262 | .75 | .75-1.5 | 1 | 20+ | 45-150 |

bounds. For example, what would happen if the following addition was attempted?

$$\begin{array}{r} 7346928315 \\ (+)\ 5947683219 \\ \hline 13294611534 \end{array}$$

The sum would clearly overflow the capacity of the machine and some provision should be made for such an occurrence. Since the machine is restricted to a fixed number of digits, the best response would be for this overflow to be detected and the computation stopped with a message to the user describing what has occurred.

Two methods of representing negative numbers are common. In the first, a single bit is employed to indicate whether the number is positive (usually 0) or negative (1). For obvious reasons this is called a signed magnitude representation. This scheme involves some ambiguity in that both +0 and -0 are valid machine words.

The second method uses complements to represent negative numbers. In a number system of radix r, the (r-1)'s complement of a digit, d, is defined as (r-1) - d. Thus, the 9's complements of the decimal digits are given by:

| Decimal Digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 9's Complement | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

Similarly the 1's complement of the binary digit 1 is 0, and of 0 is 1. The (r-1)'s complement of a number is obtained by taking the (r-1)'s complement of each digit of that number. To obtain the r's complement of the number, 1 is added to the (r-1)'s

---

α W denotes word-oriented, C denotes character-oriented, M denotes mixed orientation. The number is the number of digits or characters per word.
a denotes alphanumeric, b denotes binary, d denotes decimal.

β Memory size is given in thousands of characters or words and is core memory unless otherwise indicated. Cycle or average access time is in microseconds.

γ 80 words of core memory plus 4,000 words of drum memory.

δ 60 words of core memory available in addition to 4,000 words of drum memory.

ε 1000 words of mercury delay line memory on Univac I.
10000 words of core memory on Univac II.

ζ Add times in microseconds with - indicating that special features may decrease this and + indicating that times may be above this in some cases. Multiply times run two to ten times this amount.

η Addresses indicate the number of addresses used by most instructions.

κ Number of machines indicates total manufactured in case of older machines, but includes those on order in case of the newer machines.

λ Monthly rental is in thousands of dollars and usually includes maintenance. Machines can usually be purchased for approximately fifty times the monthly rental.

τ Indicates tube machines. All others are solid state machines.

complement in the last place. Thus the 9's complement of 246 is 753, whereas its 10's complement is 754. In complement representation, the first digit of a binary computer word is naturally used as the sign. Thus, in decimal, the first digit being larger than 4 would be taken to indicate a negative number and so three decimal digits could represent the numbers from -500 to +499. As an example we consider the following addition, where the numbers are in 10's complement:

$$\begin{array}{rrrr} & 1 & 2 & 3 \\ + & 9 & 9 & 9 \\ \hline (1) & 1 & 2 & 2 \end{array}$$

In 10's complement addition, the overflow digit (1) is discarded, so that the addition of 999 is equivalent to the addition of -1 to 123. Although complement representation is not commonly used in decimal machines, it is often found in binary machines. Both 1's complement and 2's complement are used, but 2's complement is more common. In binary 2's complement representation, the first bit being a 1 indicates a negative number. Just as in 10's complement, any overflow digit in addition is discarded. For example, -(010001) is 101111 since:

$$\begin{array}{rrrrrrr} & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 1 & 0 & 1 & 1 & 1 & 1 \\ \hline (1) & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

An advantage of 2's complement representation is that zero has a unique representation since:

$$-(000000) = 111111 + 000001 = 000000$$

Using any of these methods in a fixed word length machine with an assumed radix point at the right hand end of the word, it is possible to represent all integers (positive, negative and zero) and to perform additions, subtractions and multiplications on them with complete accuracy as long as the capacity of the machine word is not exceeded. It is even possible for the machine to detect any overflow of this capacity and the user to be warned of his violation of the bounds. Division, however, presents two problems. First, division by zero is not defined, but may be easily detected, the computation stopped, and the user warned of his difficulty. Second, division of an integer by a non-zero integer does not usually produce an integer. Although computers usually produce both the quotient and the remainder in this case, most programming systems (in order to produce an integer as a result of this operation) throw away the remainder and give only the quotient as an answer. This convention turns out to be quite valuable and the remainder can still be easily obtained when needed. As an example, consider a decimal machine with 10 digits and sign in each word. Addition, subtraction, multiplication and division of integers are illustrated by:

$$\begin{array}{rr} & +0000000013 \\ (+) & +0000000142 \\ \hline & +0000000155 \end{array} \qquad \begin{array}{rr} & +0000000123 \\ (-) & -0000000321 \\ \hline & -0000000198 \end{array} \qquad \begin{array}{rr} & +0000000017 \\ (x) & -0000000003 \\ \hline & -0000000051 \end{array}$$

$$(+0000000019) \;/\; (+0000000005) = +0000000003$$

With such use of a finite number of decimal or other digits, it is not possible to exactly represent irrational numbers such as the familiar mathematical constants

$$\pi = 3.1415926536... \qquad \text{and} \qquad e = 2.7182818285...$$

(the base of the natural numbers). Even most rational numbers cannot be so represented, as the case of 1/3 = .3333333... in the decimal number system so simply illustrates; however, the specific rational numbers which can or cannot be so represented change with the base or radix used ($1/3 = .4_{12}$, i.e. base 12, whereas 1/5 = .2 (base 10) = .24972497... (base 12)). In fact, a fixed number of digits in any radix can take on only a finite number of different values. Since the real numbers, the rational numbers, and even the integers, are infinite sets, this means that computers may operate with only very small subsets of these systems. Thus, in using computers we can only represent most numbers and do the corresponding calculations approximately.

If the numbers to be approximately represented all have nearly the same magnitude, an assumed radix point may be used and the actual calculations carried out by integer arithmetic. In multiplication and division some shifting is necessary to maintain the assumed radix point and one must make sure that the bounds of significance are not violated. It is even possible to keep track of where the radix point is in each quantity used and thus scale or assume the point differently, for different quantities involved. Even this approach requires that definite bounds be known for each quantity and that the various numbers be shifted to match each other when addition or subtraction takes place or to preserve accuracy. Although these techniques are often used in computers, they involve considerable skill and care on the part of the user. It is especially difficult to represent large or small quantities where these may be mixed in computations. For example, to represent the physical constant .00000000000000000001602 which in coulombs is approximately the magnitude, e, of the charge on an electron, it would be necessary to assume the decimal point far off the left end of the computer word. Since the numbers on which most computations are performed are approximations to real numbers (which include the rationals and the integers, but are not restricted to these), a means of representing such approximations must be provided.

Most scientific computers today provide built in floating point arithmetic which allows numbers to be represented in a form similar to the so called scientific notation. Following this notation, it is possible to express any real number by a signed fraction multiplied by a power of the radix. This fraction is called the mantissa of the number and is chosen so that its first significant digit occurs immediately after the radix point. The corresponding power of the radix by which this mantissa must be multiplied is called the characteristic of the number. The characteristic of a number is an integer, and may be either positive, negative or zero. To use this representation in a machine word, it is only necessary to divide the machine word into two parts so as to represent the mantissa and the characteristic separately.

Returning to the example machine having 10 decimal digits and sign, this floating point representation can be easily illustrated as follows: Let the sign of the machine word serve as the sign of the number (i.e. the sign of the mantissa). Use the first two digits of the machine word for the characteristic and the last eight for the mantissa. In order to accommodate the sign of the characteristic, add 50 to the true characteristic to obtain the first two digits of the machine word. This code for the characteristic is called an excess 50 representation and allows the true characteristic to vary from -50 to 49 as the machine characteristic goes from 0 to 99. A floating point zero will be represented by all zeros, but note that both +0000000000 and -0000000000 may occur. Table 2.5 shows some equivalent codes for several numbers using this representation and some others to be discussed later in this section.

Addition or subtraction of floating point numbers requires that the exponents of both numbers be the same. Thus, the mantissa of the smaller number must be shifted to the right until the exponents are equal. The two mantissas are then added or subtracted and the result normalized so that the first digit of its mantissa is non-zero. For example, in subtracting +5124635281 from +5349287512, the mantissa of the first number must be shifted two places to the right so that they have the same characteristic as shown below:

$$\begin{array}{rll} & +53 & 49287512 \\ (-) & +53 & 00246352\ \ 81 \\ \hline & +53 & 49041160 \end{array}$$

The extra digits (8 and 1 in this example) may be disposed of by either chopping them off or they may be used in rounding the last digit of the result. This would produce a result of either +5349041160 or +5349041159 depending on what is done with

Table 2.5 Floating Point Representation

| Decimal Number | Scientific Notation | Decimal Floating Point | 36 Bit Binary* Floating Point |
|---|---|---|---|
| 1.0 | $.1 \times 10^{1}$ | +5110000000 | +201400000000 |
| -10.0 | $-0.1 \times 10^{2}$ | -5210000000 | -204500000000 |
| 0.5 | $.5 \times 10^{0}$ | +5050000000 | +200400000000 |
| 0.125 | $.125 \times 10^{0}$ | +5012500000 | +176400000000 |
| 0.0390625 | $.390625 \times 10^{-1}$ | +4939062500 | +174500000000 |
| 3.1415927 | $.31415927 \times 10^{1}$ | +5131415927 | +202622077325 |
| -6.2831853 | $-.62831853 \times 10^{1}$ | -5162831853 | -203622077325 |
| 2.7182818 | $.27182818 \times 10^{1}$ | +5127182818 | +202533741242 |
| | $.1602 \times 10^{-18}$ | +3216020000 | +102572205724 |
| 20! | $.24329020 \times 10^{19}$ | +6924329020 | +276416066265 |

*Expressed in octal form.

the extra digits. The normalization is illustrated in the following addition:

$$\begin{array}{lcl} +55\ 10000000 & \longrightarrow & +.10000000 \times 10^{5} \\ -54\ 10000000 & \longrightarrow & -.01000000 \times 10^{5} \\ \hline +54\ 90000000 & \longleftarrow & +.09000000 \times 10^{5} \end{array}$$

If the two numbers are quite close in magnitude it may be necessary to shift the resulting mantissa several places to normalize it and the number of significant digits in the result may be quite small. This loss of significance may cause the results of some computations to be meaningless. For example, the two numbers 234.79265 and 234.79264 may both be accurate to the nearest digit in the last place. This means that the first may be as large as $234.792655^{-}$ and the second as small as 234.792635. Thus their difference may be anywhere between zero and $.00002^{-}$. Although computation would yield .00001, this result might be almost 100% in error in spite of the fact that the numbers used were accurate to eight significant figures.

Because of chopping or rounding and loss of significance in computer calculations, the usual rules of algebra cannot be depended on in many cases. For example,

the associative law of addition does not hold for floating point operations. The following calculation illustrates this in our example machine even with rounding:

$$A = +5125400000 \quad \text{or} \quad .25400000 \times 10^{1}$$
$$B = +5985000000 \quad \text{or} \quad .85000000 \times 10^{9}$$
$$C = -5984999700 \quad \text{or} \quad -.84999700 \times 10^{9}$$
$$A + (B + C) = (+5125400000) + (+5430000000)$$
$$= +5430025400$$
$$(A + B) + C = (+5985000000) + (-5984999700)$$
$$= +5430000000$$

In this example there is not much significance left in the sum B + C and so there are problems involved other than the failure of the associate law. This does illustrate, however, that computer calculations and abstract mathematical theory are not always in agreement.

Multiplication is somewhat more straightforward in the case of floating point numbers. The exponents are simply added, the mantissas multiplied and the result normalized if necessary for example, if 200 and .03 are to be multiplied the work goes as follows:

$$\begin{array}{rcl} +5320000000 & \rightarrow & +.20000000 \times 10^{3} \\ (*)\underline{+4930000000} & \rightarrow & (*)\underline{+.30000000 \times 10^{-1}} \\ +5160000000 & \leftarrow & +.06000000 \times 10^{2} \end{array}$$

Since the characteristics are represented in the excess 50 code the machine must actually subtract 50 from their sum. Division is performed in a similar manner.

As a more practical example of the problems of maintaining significance in computer operations, consider the solution of the quadratic equation $x^2 - 10000x + 1 = 0$. By the quadratic formula the solutions are given by

$$1/2 \left[ 10^{4} \pm \sqrt{10^{8} - 4} \right]$$

and, since $10^8$ takes 9 digits, the 4 in the units place would be lost in our sample machine of 8 digits accuracy. Thus, employing this formula we would get as roots:

$$x_1 = 10000 \quad \text{and} \quad x_2 = 0$$

The root $x_1$ is very close to one of the actual roots. Although the other root is near zero, a much better approximation for this second root may be obtained by dividing the first root into the constant term of the equation. This would produce $x_2 = 10^{-4}$ which is much more satisfactory. This technique will be used later in the text to develop an algorithm to solve quadratic equations. It is useful here to

illustrate that the way in which calculations are organized may contribute to the significance of the results.

Finally, it is observed that floating point numbers are not evenly distributed over the range of the real numbers. In our example, it is not possible to represent a number greater than $10^{49}$ or less than $10^{-51}$ in magnitude. The numbers are very close together for smaller values, but at the higher end of the range they are quite far apart. For example, the two consecutive floating point numbers +9010000000 and +9010000001 actually differ by +8310000000 which is $10^{32}$. Thus one cannot get within a "country mile" of most numbers in that range.

Conversion between floating point and integer notation is often necessary. Representing the integer 123 in both forms we have:

+0000000123 and +5312300000

The conversion from the integer to floating point may be accomplished by adding (as integers) +5800000000 to the integer form, thus obtaining +5800000123 and then normalizing the result. However, this method (which is quite similar to that used in many programming languages) is only valid for integers less than $10^8$ in magnitude. Thus, only 8 digits of significance is available for integers if they are to be converted into floating point representation.

In converting from floating point to integer, any fractional part of the number must be disposed of in some manner. This fractional part may be chopped off (e.g. 1.9999999 becomes 1) or used to round the resulting integer (e.g. 1.5000001 becomes 2). Chopping is the more common method and the one used by the MAD compiler.

Although the examples given so far have assumed a decimal machine, all the discussion is applicable to numbers represented to any base, including binary. When working with a binary machine it is convenient to write words in an equivalent (see Appendix B) octal or hexadecimal (base 16) form when their bit structure must be specified in detail. For example, with a 36 bit machine the word

111 010 101 100 001 000 011 110 101 111 001 100

might be represented by 725410365714 in octal. If the first bit is being used as a sign bit, then it may be written as a sign separately and the first octal digit limited to 0, 1, 2 or 3. Thus the above word would become -325410365714. With a 36 bit word all integers less than $2^{35}$ = 34359738368 in magnitude may be represented. In binary floating point the characteristic indicates the power of two which is multiplied times the mantissa. In the IBM 7040 and 7090/94, the floating point representation is very similar to the decimal one given as an example above. The first bit is used for the sign of the number. The next eight bits are used for the

characteristic in an excess 128 code. This characteristic may therefore range from -128 to +127 in binary which is equivalent to about -38 to +38 in decimal. The remaining 27 bits are used as the mantissa which is normalized so that the first bit is always a 1. This representation of several numbers is shown in Table 2.5.

More accuracy than is available in a single word is essential for certain calculations and so multiple precision arithmetic, which uses more than one word to represent a number, may be used. A number of machines have double precision capabilities built into the hardware, but it is always possible to program a machine to do multiple precision calculations using several words to represent each number. In any case, these operations are slower than the single precision which we have discussed and should be used only when analysis proves them necessary. Using great precision does not in itself prevent any of the computational problems we have mentioned. Although it may help, multiple precision is no substitute for good analysis and proper organization of the algorithm.

In character-oriented machines, a variable number of digits may be used for a number, but even here the memory itself has some finite size and so most numbers could not be represented exactly even if the whole memory was devoted to them. In practice, numbers must be much shorter than the size of memory and in fact for some operations, especially floating point, they are often restricted to fixed lengths.

Exercises

1. Represent each of the following integers in decimal signed magnitude form using six digits plus sign:

   a) -437 b) -32 c) +76 d) -37 e) -94

2. Represent each of the following integers in both 9's complement and 10's complement form using eight decimal digits:

   a) -1490 b) +2136 c) -48620 d) -34796 e) -964380

3. Convert the following numbers, which are represented in 9's complement form, to normal mathematical form, recalling that the first digit includes the sign.

   a) 746321 b) 843296 c) 432615 d) 916432 3) 654321

4. Find the negatives of the following numbers, which are represented in 10's complement form, and convert these negatives to normal notation:

5. a) 123456 b) 432751 c) 643257 d) 184692 e) 346819

5. Perform the following operations on the 10's complement numbers. Subtract by complementing and adding. Convert the operands and results to check your answers.

   a) 123456 (+) 684321

   b) 724369 (-) 316948

   c) 469213 (-) 824693

6. Convert each of the following octal integers to binary signed magnitude form using 24 bits (including sign) represented octally:

   a) -213  b) +463  c) -146  d) +1241  e) -2076

7. Convert each of the following octal integers to both binary 1's complement and 2's complement binary using 24 bits (including sign) represented octally:

   a) -123  b) +217  c) -244  d) +73  e) -1076

8. Perform the following operations on the octal representations of the 18 bit 2's complement numbers, converting and checking in decimal:

   a) 214763 (+) 743152  b) 614325 (-) 341572  c) 125430 (-) 543216

9. Represent each of the following numbers in decimal floating point using ten digits and sign as illustrated in this section:

   a) 471.30692  b) -206943.47  c) .0046923843

10. Convert the following numbers, which are represented in floating point machine code as illustrated in this section, to normal mathematical representation:

    a) +5743216519  b) -5314362953  c) +4621576432

11. Perform the following operations on the floating point numbers two ways, using first chopping and then rounding to dispose of the extra digits:

    a) +5346921326 (+) -5121346985  b) +5214693821 (-) +4921635193  c) -4731578261 (-) +5115632179

12. Convert the following numbers to 36 bit binary floating point expressed in octal as illustrated in this section:

    a) 3/4  b) -5/8  c) 24.5  d) -13.25  e) 6.13

13. Find the normal decimal form of the following 36 bit binary floating point numbers expressed as illustrated in this section:

    a) +204510000000  b) -203740000000  c) +175400000000

14. Convert the following decimal machine integers to decimal floating point representation:

    a) +0000016432  b) -0000004321  c) +0084325691

15. Convert the following decimal floating point numbers to 10 digit machine integers, chopping off any fractional part:

    a) +5346921815  b) -5555555555  c) +4682156021

16. Convert the following 36 bit binary floating numbers to octally expressed binary machine integers:

    a) +207463000000  b) -204542000000  c) +216453214600

## 2.5 Input and Output

Although there are many methods by which data and instructions may be given to computers, they must be prepared in a form which can be read by the particular computer being used. The most frequently used method involves putting the data and instructions into punched cards which can then be read directly into the computer. Many machines, especially smaller ones, read their data from punched paper tape, but cards are more commonly used and have many advantages over paper tape. Most cards contain only 80 or 90 characters and so another card may be punched with corrections or modifications and inserted in a deck of cards to replace one in error. New cards may also be inserted easily and card decks rearranged as desired.

In the most commonly used cards, a character may be represented in each of eighty columns by one, two or three out of a possible twelve rectangular punches. The columns are numbered from left to right by 1 through 80 and the punch rows are designated as the 12(+), 11(-), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 punches from top to bottom on the card. The characters may also be printed along the top edge of the card so that people can read the cards more easily, but this printing, along with any other printing on them and the color of the cards, is meaningless to the computer. These cards are 7 3/8 inches long by 3 1/4 inches wide and about .007 inch thick. They now usually have all their corners rounded except one which is chopped off to help in orienting the cards of a deck into the same position. In most systems, the last eight columns of the cards are reserved for identification and sequence numbering, and so only 72 columns are available to the user for data and programs. The use of identification punches helps the user keep account of the various decks he may be using. Sequence numbering of decks is a great help in locating a particular card in a long deck or in reconstructing a deck which has been dropped. Most computer programming systems today use 47 characters plus the blank which are coded as follows: A single punch indicates exactly the plus, minus or digit used to designate that row. The 26 letters and the slash "/" are represented by two punches, one called the zone punch being in the +, - or 0 row and the other being in rows 1 through 9, which uses all 27 such combinations. The special characters = , . ' $ * ( ) are punched with two or three punches, one being an 8 punch and another being either a 3 or 4 punch. Figure 2.4 shows such a card using all these codings and Table 2.3 contains a column listing all these card codings.

Computers can usually produce output in any form they can read, such as punched cards or punched tape. In addition they can usually produce printed listings using at least their input character set and often larger character sets, although they

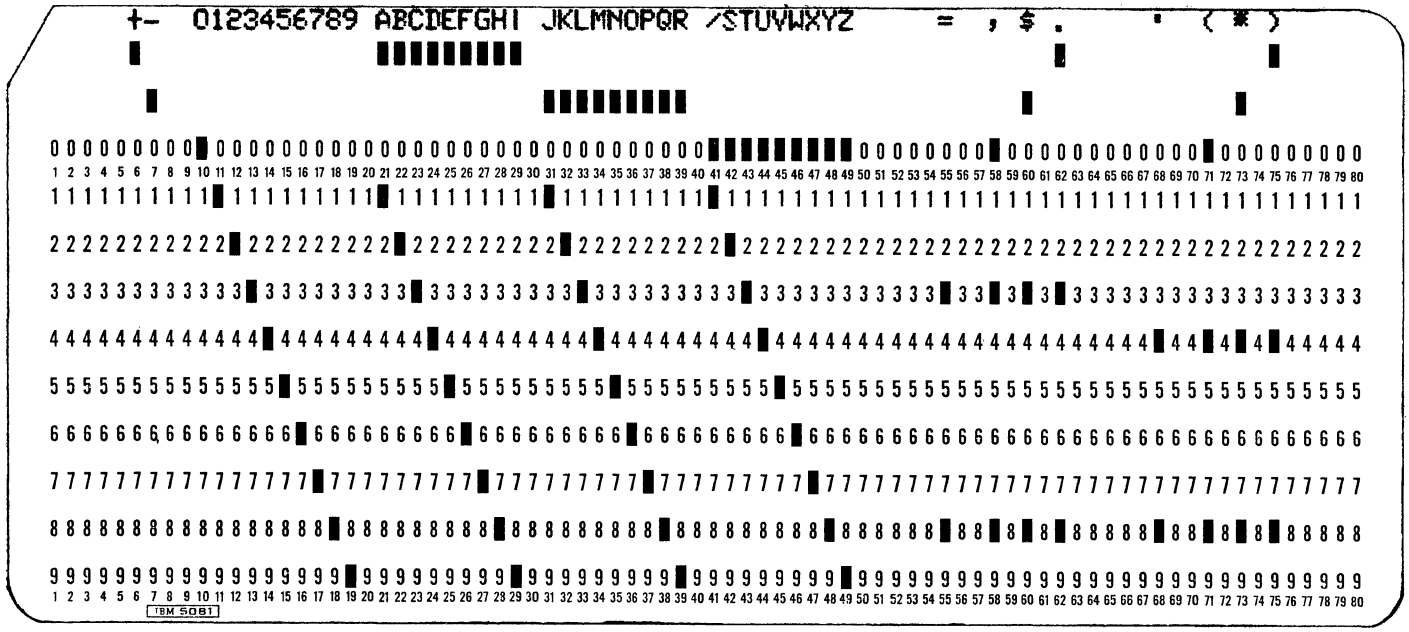



Figure 2.4 A punched card with sample punchings

are usually not able to read such printing. Printing may be done by typewriters or similar devices which print one character at a time or by line printers which essentially print a whole line at a time. Both of these devices print mechanically by bringing a type character, an ink or carbon ribbon, and the paper together. The line printers are much faster and cheaper to use if there is a sufficient volume of printing. A number of even faster printing devices which use electrostatic or photographic techniques are also in use or under development.

Cards are usually read by machines at a rate of somewhere between 15 and 2000 cards per minute. Punching cards naturally takes more time than reading them and usually occurs at 15 to 400 cards per minute. Typewriters are normally able to type between 300 and 1000 characters per minute under computer control. Line printers usually print between 120 and 136 characters per line and between 150 and 1300 lines per minute. Since these rates are all very slow relative to the internal processing speeds of large computers, it is very common to use a small computer to read cards and write the information on magnetic tapes which can then be switched to the larger machine. Similarly, these same small computers can take the output on magnetic tape from the larger machine and print listings or punch cards from it as desired. This technique increases the overall efficiency of the system.

As discussed earlier, there is great need to have more characters available for both input and output with computers. For example, one is usually restricted to upper case letters. If it is desired to operate on textual material, both upper case and lower case are needed for clarity and consistency with common usage. Such items as question marks, quotation marks, colons, semicolons, ampersands, percent signs, underlining, and many others are also most necessary in many uses. In mathematical and other technical areas the square brackets [ and ], absolute value bar |, superscripts, subscripts and many other things could be used effectively. Although the industry is gradually moving toward the provision of such capabilities, it will be some time before more convenient systems are commonly available.

The use of basic source documents which can be machine processed is becoming more common. For example: most checks now have bank and account numbers printed on them in magnetic ink; names and account numbers printed from credit cards are often machine readable; multiple choice tests and other documents use mark-sensing where the space between two dotted lines is filled in with a pencil; machines are now available which can read a printed page meeting certain specifications; and many other such schemes are available or under development.

Scientific experiments and other processes are now often designed so that they automatically record data in a form ready for the computing machine and, in fact, they are sometimes controlled directly by computers which can record the data and immediately begin computing on it. In some cases, e.g. control of space vehicles, computations must be completed, decisions made and signals returned to the experiment or process within a few milliseconds after receipt of the input data. This latter type of computing is often called <u>real-time computation</u>.

Another important means of communicating with machines is the remote station. Here the person presses buttons or keys to input data to the computer and the computer types, displays or otherwise transmits messages back to the person. In some systems under development, a person simply dials a telephone number from any telephone to contact the computer and then dials certain data such as a customer account number. The computer is then able to respond with recorded voice messages, such as statements on the status of the customer's account. In other applications the person uses an automatic typewriter or other more sophisticated device on which he types messages directly to the computer system. The system then types or otherwise displays messages to the user which allows direct interaction or <u>conversation</u> between the user and the machine. Further developments in this direction are being made and, in fact, it may soon be possible to simply subscribe to a computer service much as one does now in the case of telephones.

## 2.6 Memory and Storage

As discussed in Section 2.3, the memory and storage devices used in computing systems are basically binary in nature with some number of bits being treated together as either a word or a character. Memory and storage units are generally classified according to their type, size and access time. The size of a unit is given in terms of the number of words or characters it can hold. The time necessary to obtain a word or character from the device is usually called its access time. A random-access memory or storage is one for which the access time is essentially independent of the location in the device of the word or character to be obtained. The time necessary to store a word or character in a memory or storage is usually about the same as its access time. Theoretically, memories and stores are designed so that information can be read from them as many times as desired without destroying it; however, there are limitations on the mechanical life of tape and other equipment. With this background, we shall first discuss three main classes of memory and storage in a system, and then several types of actual devices which may be used.

The central or main memory of a computing system usually serves as the focus of the system. It is here that operands and instructions are located when they are used by the processing units and it is here that results are placed, at least temporarily. It is natural that the central memory is the fastest--has the smallest access time--of any in the system and is nearly always random-access. The main processing units, peripheral processors and input-output controllers work out of the central memory in most cases. We will use the word memory mainly in the sense of central or main memory which may be thought of as a subclass of storage in general. Since the fast and highly random-access devices used for central memories are also expensive, it is not economical to provide sufficient storage for a modern system in this form. It is therefore customary to provide additional slower storage facilities which cost correspondingly less per bit. By auxiliary storage we will mean the next fastest storage in the system other than main memory. It is used to permanently store system programs and data, and to temporarily store data after input, before output or between processes. The term supplementary storage will be used to represent even slower storage which is used for input and output of masses of data or storage of data and programs which are needed less frequently.

Most central memories in use or planned today are constructed from magnetic cores and thus are often called the core or core memory. An individual core is a small torus or doughnut made of a ferro-magnetic material and may be anywhere from

15 mils (thousandths of an inch) up to 50 mils in diameter. These cores are capable of being magnetized in one of two directions and may therefore be used to represent the two binary states 0 and 1. An array of these cores is strung on wires through which it is possible to either sense or set the direction of magnetization in each of the cores. An individual core with the wires it is strung on is shown in Figure 2.5. Central core memories usually vary in size from 40,000 to more than 4,000,000 bits. In most core memories the information in a set of cores is destroyed as it is being read into a temporary register, but it is automatically rewritten from this register immediately. A cycle in such a memory consists of the time necessary to read into the temporary register and rewrite the cores. The read or access portion of a cycle is about 40% of the total time and the word is available for transmittal to other units as soon as it is read into the temporary register. The rewrite phase of the cycle must be completed, however, before another reading operation may be started. The cycle time on core memories available today varies from 60 μs. (micro or millionths of a second) down to less than 1 μs. Core memories cost from less than 2¢ up to more than 30¢ per bit depending on the size and cycle time of the memory, but the costs are going down rapidly.

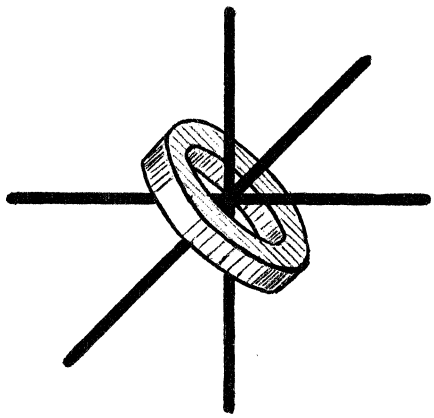


Figure 2.5 A magnetic core

Each of the words of central memory is assigned a number or address and these addresses are always used in referring to that word. The first (smallest or sometimes called lowest) address is normally zero (0) and the last (or highest) one is then one less than the size of memory. Blocks of memory are usually allocated by the system for various routines and data. For example, the systems programs which control the overall behavior of the system and perform other service functions, such as accounting for usage, often occupy a block of memory at the lower end (i.e. with the small addresses). Users programs, library programs needed by users programs and data storage occupy other areas. The allocation of memory often works from both ends leaving any unused space in the middle. Figure 2.6 illustrates such a memory allocation. Of course, in more complex systems where several processors are sharing a common memory, the allocation scheme becomes more intricate and many different methods are being employed to carry out this function.

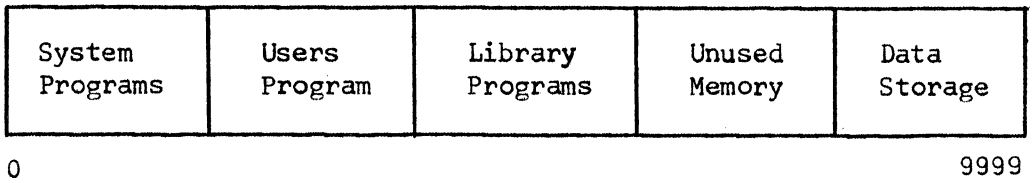



Figure 2.6 An illustration of memory allocation

In some systems the main memory is some slower device than magnetic core. Ultrasonic delay lines wherein electrical impulses are converted to ultrasonic waves in a column of mercury and electrostic memories which store information on the face of tubes similar to television tubes were used in some early machines as main memories. Magnetic drums (which will be discussed below) have also served as main memories in some systems. Auxiliary or supplementary stores take many forms, one of which may be simply more core memory which has a slower speed. Others make use of magnetic drums, magnetic disks, mechanically selected magnetic cards or tapes, and magnetic tapes which must be hand mounted on tape drives. Almost all auxiliary and supplementary storage currently available involves the recording of one of two states on a magnetically sensitive surface much as is used in audio and video tape recorders. To write information on or read information from these surfaces, it is necessary to bring a small device called a read-write head near the surface which is usually being moved rapidly. These stores differ in the type of base to which this surface is attached and the mechanical methods used to bring the particular section of surface and a read-write head together.

A magnetic drum storage consists of a cylinder which is covered with a ferrous oxide coating and rotated at very high speed. The read-write heads are located near the surface of the drum and sometimes may be mechanically moved to select different "tracks" around the perimeter of the drum. A magnetic disk storage is similar to a drum except that a number of circular plates or disks are mounted on a central shaft and rotated together. The early disk units had a single read-write head which was moved up and down the disk stack and then in between the disks. More recently, separate read-write heads for each disk surface are mounted on a common "comb" or even separate heads for each track on each disk are used. Both disks and drums are usually capable of storing several million characters, but in using them it is necessary to wait until the desired data rotates under the read head before it can be read. Similarly one must find the place where a block of data is to be written.

After this initial delay, however, a properly stored block of data can be transferred to another unit (e.g. main memory) at quite high speeds. In general drums rotate faster than disks, but disks have a greater storage capacity. Disk units are now available from which the disks themselves may be easily dismounted and moved from one unit to another. The maximum access time for drums usually lies in the range 8 ms. to 100 ms. (milli or thousandths of a second) after which data is transferred at some 50,000 to 250,000 characters per second. Maximum disk access times lie in the range 20 ms. to 400 ms. and disks transfer data at 20,000 to 100,000 characters per second. Drum memories may be purchased at about 1 cent per bit, whereas disk memories cost less than .1 of a cent per bit.

Storage systems have been developed recently which use a card or sheet of material covered with a ferrous oxide and mount many such cards in a cartridge. Several of these cartridges are mounted in a unit in such a way that the device is capable of mechanically selecting a cartridge and then bringing a particular card into the proper position for reading and writing. Although the access time is certainly slower for this type of device than for drums and disks, the greater amount of storage and its flexibility are most attractive.

By far the greatest present source of storage, and in many systems the only one used other than core memory, is magnetic tape. The most commonly used tape is 1/2 inch wide and comes in rolls of 2400 feet mounted on a 10 1/2 inch reel. Since the desired data may be far down the tape from where it is positioned under the read-write head, tape is not random-access. In spite of this handicap, tapes can be used very effectively in many applications. Often the work must be very carefully laid out with the limitations of the tapes in mind. The extreme portability and low cost of tapes have made them most valuable so that any major computing installation has a large library of them on hand. Tapes are especially useful for input-output, back up storage, storing large volumes of seldom used data, long term maintenance of records, and transfer of data from one installation to another. Data is packed on tape at from 200 to over 800 characters per inch and so a single tape may hold millions of characters of data. Although the time necessary to wind a tape from one end to the other may vary from less than one minute to more than 10 minutes, after information is once located on the tape it may commonly be transferred at rates in the range of 6,000 to 60,000 characters per second.

Each memory or storage unit must incorporate certain control functions in its operation since it may receive different types of requests from the processors or controllers. One such type of request is for a copy of some data which is referred to by name, address or other designation. A second is a request to keep or store

some data at a specific location or under some designation. A third type involves special functions such as selecting a particular section or unit of storage, starting into motion or rewinding a tape, or inquiring as to the status of some part of the memory unit. In case of simultaneous requests from various processors attached to a memory, the memory control must establish which has the greatest priority or need. Since all the processors must reference it, the central memory, in performing this later function, sometimes assumes the highest level of control in the system.

More recently, memories have been built using thin films of material deposited in a vacuum on a glass substrate. Other techniques which promise to be faster, more compact and less expensive are also under development. In addition, a number of different ways of organizing the structure of memory and storage have been proposed, partially developed and in some cases given limited use. Some of these which do not use addresses in the way that has been indicated, but refer to data by part of its contents, are called content addressed memories (CAM's) or associative memories. Others allow complex cross references to data in a hierarchial structure and may, for example, be called list or trie memories. Thus many new techniques of both constructing and organizing storage and memory devices are now being developed and should be in use soon.

## 2.7 Processing and Sequential Control

A processor or processing unit is a section of a computer in which operations are performed on data in accordance with instructions taken from the memory. Usually, the data are obtained from memory, storage or input devices and results are sent to memory, storage or output devices; although many processors also have the ability to store some data in high speed registers. Such high speed registers are constructed of flip-flops made with transistors or diodes, and operate much faster than memory. A central or main processor nearly always contains a number of these registers and operates primarily on data stored in central or main memory and these registers. Very often one operand is taken from memory and the other from a register with the result being returned to a register. Although some processors may perform a wide variety of operations, others are designed for some specific purpose. For example, input-output controllers can only transfer data between memory, input-output devices, or auxiliary storage devices.

Each processor has a set of operations which it is able to perform and a corresponding set of instructions which cause it to perform these operations. An instruction consists of an operation code identifying the operation to be performed,

perhaps one or more references to data or instructions in registers or memory, and sometimes other information. Sometimes the references to registers or memory is implicit in the operation code, especially in the case of registers, but addresses for the memory cells (and registers in more modern machines) must be included in most instructions. Some instructions use two or even three addresses so that machines are often characterized as one-address, two-address or three-address machines, depending on the most commonly used type of instruction. Some instructions cause an operation on a block of words from memory by referencing its beginning and end or its beginning and a count of the words affected. In other cases, instructions use part of the data itself to determine where they stop. These latter features are especially common for operations which simply move data in memory or for input-output operations.

To make the following discussion more simple, only symbolic addresses and operation codes will be used. The address of a particular location in memory will be denoted by N and its contents by (N). Mnemonic operation codes consisting of three letters, which suggest the name of the operation, will be used. This type of notation for instructions is called symbolic assembly language and it can be translated to actual machine instructions by a machine using a program called an assembler. Only instructions for a one-address word-oriented machine will be discussed in detail and, in particular, the specific instructions and registers used as examples in this discussion are those found in the IBM 7040 and in the IBM 7090.

A common arrangment in the central processors of word-oriented machines has been to use two main data or arithmetic registers called the accumulator (AC) and the multiplier-quotient (MQ) registers, plus some temporary registers and some special purpose registers which will be discussed later. Computers of recent design usually provide many such registers and the tendency is to make these registers usable for general purposes. An alternative arrangement, used frequently in the less expensive machine, is to assign certain locations in memory to serve as some of the registers. In character-oriented machines, the registers usually hold only a character apiece, but the individual instructions in such machines often cause it to operate serially on several characters from memory and return the results to memory one character at a time. For this reason instructions for character-oriented machines often use two memory addresses; one denoting an operand and the other denoting a second operand to which the result is also returned. An instruction may not involve memory at all, but work only on the registers of the processor. Examples of such instructions are: XCA meaning "exchange (AC) and (MQ)" and CHS meaning "change the sign of AC".

A set of instructions which has been coded in the form of numbers or other symbols is stored in memory and the processor is initially set to execute or carry out a particular one of these instructions. Most processors normally execute the instructions in the same order that they are stored in memory, although some instructions may indicate another instruction which is to be performed next after their execution has been completed. Such processors usually have a control or location register (CR) which indicates where the next instruction is to be obtained. When an instruction is taken from memory this control register is increased or incremented by one so that unless that instruction modifies the CR again, the next instruction will come from the next location in memory. An instruction which may modify the CR is called a transfer or jump instruction because it changes the location in memory from which the instructions are being taken. As discussed earlier, a modern computer can take alternate paths depending on the data and other conditions which the program can test. Instructions for this purpose are called conditional transfers or test instructions. Examples of such instructions are: TRA N meaning "put (N) into CR and thus take the next instruction from N", and TPL N meaning "transfer to N if (AC) $\geq$ 0, but otherwise do not modify the CR and thus take the next instruction in sequence.

In word-oriented machines, a very necessary processor operation is the fetching of a copy of some data from memory into a register, often called loading the register or sometimes moving data into the register. These operations only copy from memory; the original contents of memory remain intact. Similarly, the putting or storing of a copy of a register or some special value into memory is quite a fundamental operation. Examples of such instructions are: LDQ N meaning "load MQ with (N)", STQ N meaning "store (MQ) in N", STO N meaning "store (AC) in N", and STZ N meaning "store zero in N".

The simplest arithmetic operation in word-oriented machines is the addition of a number to one of the registers. Such a register may be used to accumulate sums; hence, the name "accumulator". Since addition takes so little time, it is often economical to load an accumulator by first clearing it to all zeros and then adding the number from memory. Subtraction is accomplished by simply adding the negative of the number. Examples of such instructions are: CLA N meaning "clear AC and ADD N" or "put (N) into AC", ADD N meaning "add (N) to (AC) and put sum into AC", and SUB N meaning "put (AC) minus (N) into AC".

In multiplying integers the multiplier is loaded into the MQ register, the multiplicand comes from memory and the result, being twice as long as the operands, goes into the combined AC-MQ register. Conversely, division for integers or fixed

point numbers uses a double length dividend occupying both registers AC and MQ, takes the divisor from memory, and puts the quotient in the MQ register and the remainder in the AC register. Mnemonics for these operations are MPY N and DVP N. The details of these operations are so involved and highly machine dependent that we will not discuss them further.

Floating point arithmetic operations are usually also available on scientific machines. Instructions for them are similar to the others listed above except that the arithmetic is carried out in floating point. Examples are: FAD N meaning "floating add (N) to (AC) and put sum in AC", FMP N meaning "floating multiply (N) by (MQ) and put result in AC", and FDP N meaning "floating divide (AC) by (N) putting the quotient in (MQ)".

The time necessary to carry out the various instructions varies considerably. First, the instruction itself must be brought out of memory and decoded. Although execution of some instructions can be completed in the memory cycle time used to access that instruction, as for example, with register operations and transfer instructions, most instructions require an operand which must be obtained from memory. Thus, most instructions take at least two memory cycle times. Integer addition, storing and fetching can all be done within these two cycle times; however, some other operations like multiplication, division and floating point addition take several extra cycle times on most machines. Although many of the slower computers can perform only a few thousand operations per second, the faster ones can execute well over one million operations per second when performing certain types of calculations.

| Label | Operation | Address | Remark |
|---|---|---|---|
| L | CLA | A | (AC) = (A) |
| | ADD | B | (AC) = (A) + (B) |
| | LDQ | A | (MQ) = (A) |
| | XCA | | (AC) = (A), (MQ) = (A) + (B) |
| | SUB | B | (AC) = (A) - (B), (MQ) = (A) + (B) |
| | TPL | M | If (AC) $\geq$ 0, TRANSFER TO M |
| | CHS | | (AC) = -(AC) |
| | STZ | A | (A) = 0 |
| | TRA | N | TRANSFER TO N, i.e. (CR) = N |
| M | STQ | B | (B) = (A) + (B) |
| N | STO | C | (C) = $\lvert$(A) - (B)$\rvert$ |

Figure 2.7 A sample of symbolic machine code

Some of the instructions which have been given as examples above will now be used to write a short segment of symbolic assembly language program. This program, shown in Figure 2.7, causes two of the three memory locations named A, B and C to be given new values. The variable C is always set equal to $|(A) - (B)|$. If $A \geq B$, B is replaced by A + B; whereas, if $A < B$, A is set equal to zero. It is assumed that execution begins with the instruction labeled L at the left, and that M and N are the addresses where the instructions with those labels are stored. The new values which this program produces are shown in Table 2.6 in a form often called a decision table. These tables may list actions taken under various conditions in columns op-

Table 2.6 A simple decision table

| Variable Name | Value assigned under condition | |
|---|---|---|
| | $A \geq B$ | $A < B$ |
| A | A | 0 |
| B | A+B | B |
| C | \|A-B\| | \|A-B\| |

posite the quantities they affect. Such tables often may be used to clarify algorithms and in some applications have advantages over flow charts and other notations.

Some special registers called index registers are often used to modify addresses and thus select an element from a block of memory. For example, the value in this register may represent the subscript of a vector. These registers, which are usually just large enough to hold any address for the memory, contain numbers to be added to or subtracted from the address given in the instruction to obtain an effective address which is then used to locate the operand of the instruction. There are numerous instructions available to load, modify and test these registers. Index registers are also useful for counting, controlling loops and many other things. Many of the newer machines with general purpose registers use them for indexing.

In addition to the operations mentioned above, most computers can perform operations involving logical or Boolean operations on a bit by bit basis, comparisons, shifting of words in the registers and many other things which make it possible to efficiently compute with and manipulate data.

Central or main processors available today usually involve more sophisticated features than have been discussed here. Registers are sometimes linked with memory into a push-down stack consisting of a vector in which one can store elements at the top end or from which one can fetch the top most element no matter how many elements are in it. Various schemes of indirect reference to memory, where the instruction address indicates where the address of the operand, rather than the operand itself, may be found, are often available and can increase a machine's efficiency. The processor may use look ahead features to access instructions and their data in advance of their use, and thus speed up operation of the machine. There are many other such features available in various machines.

Special input-output controllers have been developing along many lines. Often they are tailor made to transfer data between input-output devices and storage, or between various storage and memory devices. Such controllers usually have almost no arithmetic or other manipulative capacity. Some manufacturers, however, are tending to use small ancillary computers having their own memory for these purposes. This approach does provide a great deal of flexibility since this ancillary computer can be programmed, store its own instructions and temporarily store the data being transferred.

Exercises

1. Write symbolic assembly language instructions to compute the following expressions in floating point and store the results in a variable X:
   a) B * (C-D) + E/F*(-G)
   b) (R+S)/W - U*(-V)*S
   c) -U*V/M + (P-Q)*R
2. Write symbolic assembly language instructions to find the largest (algebraically) of three integers A, B, and C and store its value in the variable M.
3. Write symbolic assembly language instructions to carry out the algorithm of Figure 1.2. Use RDD A,B to read data, PRT "Anything" for the print statements and SYS to return to system.
4. Write symbolic assembly language instructions to carry out the algorithm of Figure 1.8b using the notation of problem 3. Watch the difference between integer and floating point numbers.

## 2.8 Software and the MAD Compiler

So far this discussion of machines has been devoted mainly to the hardware of machines, that is, the actual machinery, cards, tapes and their physical attributes. A second aspect of machines and their use which has now grown to at least equal importance is called software and consists of a collection of programs for the machine which are designed to make the use of the computer easier for the individual or more efficient for the organization.

Programs were first written in a very awkward language often using binary, octal or hexadecimal numbers and invariably involving much detailed clerical work in their construction. In fact the basic machine language of even today's computers is a detailed one involving many trivial but important decisions in the construction of a program. Early in the development of modern day computers it was realized that the machines themselves were more capable than people of doing much of the detailed work involved in writing programs and that man could thus be freed from this work. As a result, hundreds of languages have been developed which make it easier for people to give instructions to computers. Some of these languages, called assembly languages, are very close to the machine language of the particular machine, but allow the programmer to use decimal addresses and constants, mnemonic operation codes such as ADD, and names of the locations of variables or constants such as X, SAM and T4. This was the form in which we expressed machine language in Section 2.7. More sophisticated languages allow the use of statements which cause several machine instructions to be generated. Some of the simpler of these advanced languages are called macro or autocoder languages and others having a more sophisticated capability are called procedure or problem oriented languages. These latter languages, which allow statements like X = A+B+C, are each designed to handle programs or algorithms for a certain problem area. Some of these languages (e.g. FORTRAN, ALGOL, MAD and JOVIAL) are designed primarily for mathematical or scientific type problems, whereas others (e.g. COBOL) are oriented toward commercial applications. The new programming language (PL/I) of IBM is designed to serve both the scientific and commercial areas. Some languages for even more special problem areas are: SIMSCRIPT and GPSS, both designed for simulation problems; IPL-V, LISP and COMIT, which are designed for list processing; and SNOBOL, which is designed for string manipulations. Although these areas may not yet be familiar to the reader, they are mentioned to illustrate the wide variety found among the hundreds of languages in use today.

If the user is going to write his directions to the computer in these languages

(called source languages) which are not natural to the computer, the computer first must have been instructed as to how to convert the statements and declarations of these languages into instructions which it can carry out. There are two general approaches to this conversion. The first involves programs which interpret the users' source language as it is used by the machine. If a statement is used more than once it must be interpreted each time, and thus a large proportion of the time used for solution of a problem may be devoted to this conversion. In the second method, the users' source language is actually translated once and for all into the language of the machine, called the object language, and the machine then works from these translated instructions. Both techniques have their advantages and the same user language may often be either translated or used interpretively. Translators for assembly languages are usually called assemblers and those for procedure languages are often called compilers. In either case someone must prepare these interpreters and translators, and make them available in the computer system before the user can present instructions to the computer in the corresponding language. These converters are all part of what is now called software.

Independent of the language used on a computer, a routine developed by one person to solve a particular problem may be shared by arbitrarily many other users to solve that same problem on the same or similar machines. If a language acceptable to more than one machine is used the routine can be more widely shared. To make this sharing easier, programs which have general application are often carefully documented and kept in libraries for others to study and use. Such libraries exist in the following two forms: (1) the documentation which makes it possible for people to learn whether and how to use these programs; (2) the programs themselves which may be available either on cards, magnetic tape or directly built into the software of the computer system. Very basic programs to perform input-output conversions, to calculate common mathematical functions and to carry out other similar processes are always available in these libraries. To encourage the sharing of programs, most computer manufacturers help support organizations devoted to the purpose of disseminating programs and their documentation between all the installations of their equipment. Routines in these libraries are called library routines.

Given a computer system having a large set of library programs, translators and users programs stored and available to it, it must be possible to instruct the computer which ones to use at any given time. To satisfy this need, as well as some others, a rather new area of software called systems programs has emerged. These programs do the selecting and scheduling of work, the accounting for use, the basic input-output operations, the communication with consoles, and generally handle a

multitude of routine work connected with the operation of the system. The software used with a hardware system serves as its "training" and the same basic computer may be trained to behave in many different ways. Thus, in addition to knowing the model of computer and the particular detailed configuration of hardware equipment an installation has, one must also know what software is in use. It is this combination of hardware and software which is normally referred to as the computer system.

Well designed systems have the ability to detect many of the errors committed by both the user and the system. Some of these errors can actually be corrected by the system, but a good system will notify the user of such corrections. When errors are detected which the system cannot correct, it should at least provide the user with some meaningful description of or clue to the error. These descriptive statements are appropriately called the diagnostics produced by the system. Although hardware may help the system detect many of the errors, the actual diagnostic statements are built into the software. Diagnostics are built into almost all aspects of software, including translators, interpreters, library routines and systems programs.

Since the reader will probably be using the MAD language to write and test algorithms, some knowledge of how the MAD compiler works is useful. Since many properties of this compiler apply to others as well, it also serves as a good example. In particular, the information presented here is intended to help the user of MAD understand some of the diagnostic statements which the compiler will make concerning errors in his programs.

In the development of a language and its translator, several questions must be considered. How efficient will the object program be? How long does the translation itself take? How difficult is the source language to learn? What will be the primary use for the language? The answers to these and other questions shape the decisions which must be made.

The MAD language was designed for mathematical, scientific and engineering type computations, but with strong interest in logical, symbolic and other similar facilities. The use planned for MAD was primarily in an educational and research environment rather than a production environment. In these cases programs are often rewritten and generally run only once or few times after they succeed. In this environment, a high speed of translation is more valuable than object program efficiency so the MAD compiler was designed to compile very rapidly. This compiler does, however, produce about as efficient an object program as most other translators. Since thousands of students were to learn the MAD language, it was essential that this learning process be as easy as possible. One way of achieving this ease

is to make the statements as natural as possible and thus reduce the likelihood of writing incorrect statements. Actually there are still many incorrect things which one can write in MAD, but the number of trivial arbitrary rules which one must learn has been reduced as compared with FORTRAN and some other languages. Readability of the source program is also important to the user, as well as an instructor, and this is one of MAD's strong points. Simplification of some of the more detailed operations such as input and output was another goal achieved by the MAD language and compiler. Finally the production of meaningful diagnostics was given much attention in developing the compiler, the library programs and the system programs which supervise the use of MAD.

The MAD compiler consists of three phases, designated I, II and III, which will be discussed separately.

In phase I, the compiler reads each statement or declaration and identifies its type. It then decomposes the statements into the form of tables which are later used by phase III. Also in phase I a number of tables containing information about dimensioning of arrays, mode of variables and such things are accumulated from declarations and other places. These latter tables are used by phase II.

The identification of statements and declarations in phase I is relatively simple. If an =, (, . or ' does not appear in the first seven non-blank characters, the compiler checks for the initial key words of MAD statements and declarations, such as WHENEVER, PRINT RESULTS, END ØF PRØGRAM, TRANSFER TØ, INTEGER, etc. Recognition of such words identifies the type of statement or declaration which can then be checked and processed. If an apostrophe is found then the compiler checks the abbreviations of these initial key words such as W'R, etc. These abbreviations are discussed in detail later in the text. If one of the special characters = or ( is encountered then the compiler identifies the statement as a substitution statement and attempts to process it on that basis. If a period is encountered the compiler assumes the statement to be a subroutine call (which we will discuss later in the text) and processes it on that basis. Since identifiers may have at most 6 characters, all legal statements will fall into these four categories.

After a statement has been identified as to type, the translator checks the form of the statement as it decomposes it into elementary operations. To illustrate this decomposition process we will consider the problem of translating the simple arithmetic assignment statement

```
X = -A * B + (C - D)
```

This statement is passed on to phase III in the form of the elementary operations

$$T_1 = C - D$$

$$T_2 = -A$$

$$T_3 = T_2 * B$$

$$T_4 = T_3 + T_1$$

$$X = T_4$$

Here the $T_i$ indicate temporary storage locations which may be registers or memory cells, depending on decisions which must be made in phase III.

During phase I a listing of the source program is prepared along with diagnostics concerning the individual statements and declarations. The statements are numbered at the right in this listing which helps identify them for diagnostics produced by phase III. When a statement or declaration is found to be in error by phase I, it is not compiled and a phase I diagnostic is printed immediately underneath it in most cases. Since the first error causes work on the statement or declaration to stop, there will be at most one diagnostic after each statement. Thus succeeding errors in that statement may not be detected until after the earlier ones have been corrected and the program recompiled. The deletion of incorrect statements or declarations may cause other diagnostics to appear on later statements which are themselves correct. Similarly, some errors in later statements and declarations may be masked by earlier errors. Thus the program and its error messages must be studied as a whole, taking into account any interactions which might occur. If one or more fatal errors (i.e. errors which will prevent a complete compilation) are encountered in phase I, then phases II and III are not carried out; however, all statements and declarations are checked by phase I.

An illustration of some phase I diagnostics is given in Figure 2.8. The first statement shown in that figure was supposed to be a READ AND PRINT DATA statement, but punching was started in column 11 rather than column 12. Since the first character of the statement is R and an R in column 11 signals a remark, this statement is treated as a remark and ignored by the compiler. The fact that the R is not printed and the rest of the statement is displaced three characters to the left in the listing should clearly indicate to the user that something is amiss. In the second statement (which is here labeled *001 because of the error in the first statement) a multiplication sign has been omitted between the 2 and the X, but how would the compiler know whether the user meant multiplication, addition or some other operation? All the compiler can determine is that two quantities are side by side with

```
         S     EAD AND PRINT DATA     X, A(0), A(1), A(2)
                S = 2 X                                                  *001
***** ERROR 37337 .... MISSING OPERATION OR PUNCTUATION
                Y = A(0) + (A(1)+A(2)*S *S                               *002
***** ERROR 25533 .... ILLEGAL () OR $ STRUCTURE.
                TRANSFER TO S                                            *003
                PRINT RESULT  Y                                          *004
              E D OF PROGRAM
***** ERROR 25416 .... ILLEGAL CHARACTER IN COLUMN 11.
***** ERROR 33657 .... NO WAY TO REACH THIS STATEMENT.
***** ERROR 25511 .... INCOMPLETE PROGRAM.
```

Figure 2.8 Sample phase I diagnostics

no intervening operation or punctuation. In the statement numbered *002, the parentheses do not pair off evenly and the compiler reacts appropriately. The TRANSFER TO S statement is correct in itself, but should have been placed after the PRINT RESULT statement. In its present location, the PRINT RESULT statement cannot be reached during execution of the program. The error comment on this situation is unfortunately delayed until after the next statement because that next statement is misplaced on the card and therefore incomprehensible to the compiler. Since END OF PROGRAM was punched beginning in column 10 the N falls in column 11. An N in column 11 being illegal, the statement cannot be processed and is therefore treated as a remark but with an error message. Finally, the computer finds no legal END OF PROGRAM statement and comments on the fact that this makes the program incomplete. The error numbers which appear in the diagnostics are not meaningful to the normal user. They refer to internal locations in the MAD compiler and are helpful mainly to those working on the compiler itself.

Thus in the example of Figure 2.8, only six statements contained five errors which caused seven phase I signals of trouble; however, there are still errors to be found in this program. On correction of the five errors and rerun of the program the PRINT RESULT Y statement will be followed by the following diagnostic:

```
***** ERROR 33627 .... ILLEGAL STATEMENT TYPE OR PUNCTUATION.
```

The statement is supposed to be PRINT RESULTS Y even though there is only one result to print.

In phase II, the translator uses tables produced by phase I to allocate memory to the various variables and arrays which are used in the program. The most common diagnostic from phase II is a warning to the user that certain names have occurred

only once in his program. Since it is sometimes legal for names to occur only once, especially when they are names of variables read in by READ AND PRINT DATA or labels of statements which are not referenced, the compilation is not stopped when this message occurs. This list should always be carefully checked, however, since it often gives a clue to some serious error in a program. There are other diagnostic statements which may be printed by phase II, but it is important to remember that they all concern memory allocation and will be printed after the listing of the source program has been completed. Furthermore, these phase II diagnostics will not be obtained unless the program clears phase I without the detection of fatal errors.

In phase III, the translator actually generates the object program code by putting together the tables produced by phase I for each statement with the memory allocation produced by phase II. For each arithmetic operation to be performed it checks the mode of the operands and prints a diagnostic if they are incompatible. If they are compatible, it looks up how to carry out this operation and writes the code for it. The diagnostic messages from phase III are printed after the listing of the source program is complete and after any diagnostics produced by phase II. In general, the errors which cause phase III diagnostics are more subtle and the diagnostic messages less helpful to the beginning user.

Returning to the example of Figure 2.8 and correcting the six errors already pointed out, the source program will succeed in passing all the phase I tests, but produces a diagnostic indicating possible trouble in phase II and then runs into serious trouble in phase III. The program with phase I errors corrected is shown in Figure 2.9a. The phase II diagnostic concerning the variable X which appears in Figure 2.9b does not indicate trouble since X is read in as part of the input data. The phase III diagnostics shown in Figure 2.9c indicate two errors. The first and third of the diagnostics are caused by the same error. Since the name S was used as a label on statement *001, it is declared to be in statement label mode. Thus in statements *002 and *003, the compiler has a statement label which is used as a variable. Arithmetic operations on statement labels being undefined, the compiler rejects the program with appropriate diagnostics concerning the statements *002 and *003. This error may be easily corrected by changing the statement label on statement *001 and in statement *005 to something other than S, for example SS. The second diagnostic concerns the variable A which is subscripted in the program. Variables which are subscripted are checked in phase III against the tables produced by phase II, to make sure they have been appropriately dimensioned. The statement

```
DIMENSIØN  A(2)
```

may be added to the program to correct this error. With these two corrections the

```
S       READ AND PRINT DATA     X, A(0), A(1), A(2)                    *001
        S = 2*X                                                        *002
        Y = A(0) + (A(1)+A(2)*S)*S                                     *003
        PRINT RESULTS Y                                                *004
        TRANSFER TØ S                                                  *005
        END ØF PRØGRAM                                                 *006
```

(a)

```
THE FOLLOWING NAMES HAVE OCCURRED ONLY ONCE IN THIS PROGRAM.
COMPILATION WILL CONTINUE.

     X              *002
```

(b)

```
INVALID OPERATOR-MODE COMBINATION   00540 (000/M/M) IN THE STATEMENT ENDING
                                                          S = 2*X        002

ARRAY VARIABLE   A   HAS NOT BEEN DIMENSIONED OR A CONSTANT SUBSCRIPT IS TOO LARGE.

INVALID OPERATOR-MODE COMBINATION   00304 (000/M/M) IN THE STATEMENT ENDING
                                          Y = A(0) + (A(1)+A(2)*S)*S       003
```

(c)

Figure 2.9 Sample phase II and III diagnostics

program will compile and execute properly, but will still produce the phase II warning diagnostic. Thus the phase II and III diagnostics must be studied very carefully and, in particular, the use of any variables appearing in these statements must be traced throughout the program. There may be occasions when the less experienced user must consult others more familiar with the internal workings of the MAD compiler in order to isolate certain evasive errors.

Once a MAD routine has been successfully compiled, a table showing considerable information about the object routine is printed. This table shows the total number of locations used, the relative location in the object routine of the variables and constants, a function dictionary, a statement dictionary, and some other information. The beginning user can ignore most of this information until he gains sufficient experience to take advantage of it.

The fact that a routine compiles without diagnostics does not imply that it is correct, for some hints on debugging and testing techniques, see Appendix C.

## 2.9 Using Computer Systems

Now that some general information about computer systems and their capabilities has been presented, it is worth considering how these systems may be used effectively. Since the cost for a given amount of internal computation usually decreases significantly as the power of the computer on which it is run increases (assuming efficient use of all machines), it is normally desirable to use the largest machine possible for a given application. Not many individual users can keep such a large machine busy for more than a few minutes per day, however, and so these large machines must be shared among many users if they are to be used efficiently. The use of a more powerful machine makes it possible for the individual to run more complex and sizable computations. In addition, more sophisticated software systems can be developed for the larger machines and these make the work of the computer user easier and more efficient. Since the computational needs of most organizations are growing rapidly, it is usually desirable to design or acquire systems with more capacity than is presently needed and to plan for their future expansion. To satisfy the needs of some organizations, it is necessary to have several machines which may vary in design so as to better fulfill specific computational needs. In some applications a large computer is not needed or even the most efficient. For example, input-output conversions from cards to tape and from tape to printing may be more efficiently done by smaller machines in many cases. In other applications, it is not wise to rely on a single computer whose need for repairs would leave no computer capacity available at times when it is vitally needed. Thus, there are many matters to be considered in trying to set up and maintain an efficient and convenient computer system.

There are several ways in which the effectiveness of the use of a computer system can be measured. One such measure is the volume of effective work which can be put through the system, now often called the through-put. From the point of view of a user, a very important factor is the length of time from his submission of a program to be run on the computer until the return of the results of that run. This time is commonly called the turn-around time for the system. If good through-put is achieved at a sacrifice in turn-around time, the computer may be used efficiently, but with an uneconomical use of the people involved. Thus, the effectiveness of computer usage should not be considered independently from the people who use the computer. Measurements of this overall effectiveness are more difficult to make and are usually more subjective than objective.

In most systems today, the computer has no means of direct communication

outside the room where it is installed and the individual users need nearly all the facilities of the machine during the time their problems are being run. Under this arrangement other users must wait until the machine is available to do their work and thus a queue of users, jobs, or problems waiting to be run on the computer develops. The user may have to wait several hours to get his job onto the machine and obtain his results. Since most users have other things to do, they must usually leave their work to be run by the operators of the machine and return later for the results. A messenger may be hired to carry work back and forth in order to save the users time. In order to give the users some relief from delays caused by such a mode of operation, long jobs are usually delayed until the wee hours of morning and priority is given to very short runs, especially those involving the testing of programs. Although this helps the situation somewhat, such a system normally has a turn-around time measured at least in hours and so the service provided is not as good as could be desired.

In many of these systems (including most of those where MAD is used) the user submits a deck of cards for a computer run or job. This deck consists of a program, possibly followed by data. If large amounts of data or long routines are necessary, they may already be on magnetic tapes which the user submits along with the deck of cards. These job decks are then batched into large sets by those running the computer, and often a small computer is used to load these batches onto a magnetic tape called the input tape. This tape is then switched to the large computer and the batch of jobs is run. When running each job of the batch the operators must first mount any tapes that the user has supplied. Next the user's program is read and translated into machine language, if necessary. The translation process writes information about the program on an output tape. If the user did not request that the program be run, or if fatal errors were found in the translation process, the system will then go on to the next job. If translation is successful and the user requested the program to be run, then the program is loaded into the machine by a part of the system called a loader. The program is then run or executed and uses the data supplied by the user from the input tape. The results of the run are then written on the output tape. After the batch of jobs is completed, the output tape is switched back to a small computer which lists the program information and results on paper. This listing is then returned to the user along with his deck and possibly his tapes. Such a system is called a batch processing system and is usually used on machines having no auxiliary or secondary storage other than magnetic tapes.

Several variations from the batch processing technique are currently in operation and offer a much improved relationship between man and machine. They all require that the computer system have massive storage or memory capacity with relatively low access time and, thus, usually disks or drums are employed. With such facilities, jobs may be loaded into this storage immediately upon receipt; queue and priority handling can be done by the computer system, sometimes by a smaller computer directly linked to the main computer; and results may be listed almost as quickly as they are finished without waiting for a batch of jobs to be completed. Furthermore, it is possible to locate card readers and printers remotely from the central computer facility. Thus the messenger, transportation and handling times, and many other delays and inconveniences can be eliminated. Another possible feature in such a system involves keeping the user's program in the mass storage and allowing him to simply make modifications to it as he finds errors or changes his needs. The user then has only to call for his program to be carried out and provide the parameters and data on which it is to operate. Thus, much of the time necessary to input material each time the job is run in a batch processing system may be eliminated, but must be replaced by expanded storage facilities.

Even more sophisticated systems are now in use and involve what has come to be called time sharing. In these systems the time of various parts of the computer facilities is shared in small increments among several users who communicate with the system through remote consoles. These consoles are usually similar to electric typewriters, although devices which use a display similar to a television tube are also now in use. The user types or otherwise gives messages, usually very short, to the machine through his console and the machine devotes only a fraction of a second to responding to each message. The machine then goes on responding to other consoles while the user composes his next message. This response may be so immediate that the user's thoughts and work are hardly delayed at all. Larger production jobs can be requested at any time, but their execution is delayed until times when few users are at their consoles. Less important work not involving the time of users can be run on a lower priority in between responses to the consoles. With such a system the individual statements of an algorithm or program are usually treated as separate messages and immediately checked by the computer for at least the more obvious errors. Thus typographical and most other errors may be eliminated from an algorithm in a very short time. More subtle errors may require that the user leave his console so that others may use it while he ponders his predicament. Even in this case he may leave a copy of his material in the system and return later to modify it, possibly from a different console. Only a telephone line is needed to connect

such a console with a computer system and, in fact, it is possible to dial into computers all over the country with some such consoles. Systems of this type are becoming increasingly popular and it is expected that some day almost all large computers will be operated in this way. Possibly the consoles of the future will themselves be small computers, and more simple problems such as are now done on desk calculators can be done without communicating with the larger system.

Although systems vary substantially in how they handle a request from a user, the content of such a request to these systems has some common features which are fairly independent of the system. First of all the user must usually identify himself to the system by giving his name and an account number under which a record of the computer time he uses is kept. An ID card or line on a remote console is usually used and may contain other information than what has been mentioned. Second, the user must inform the system which instruction system or language is going to be used. The user must then give the system his instructions in the appropriate language together with any parameters or data necessary to carry out these instructions. The most simple request would be for the system to apply a procedure to some data, where both of these are already contained in memory or storage. Such a command statement, illustrated by Figure 2.10, can be very brief since it must only identify the procedure, the parameter and data, and specify where the results are to go. The results might, of course, simply be left in the system. If the procedure, but not the data, is in the system, the user needs only provide the data and indicate in the request that he is doing so.

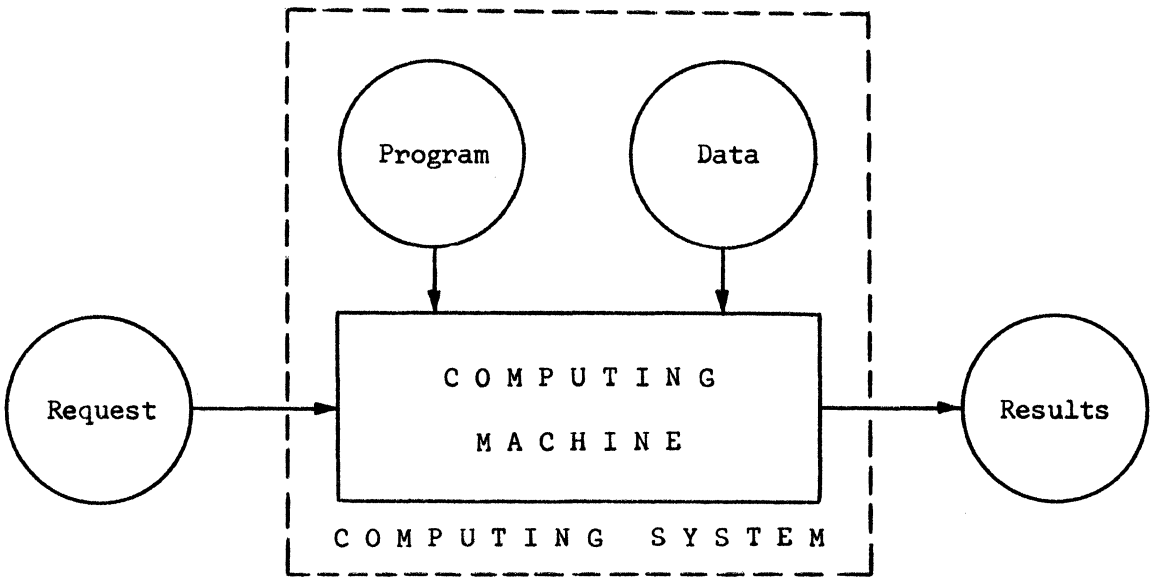



Figure 2.10 A simple request for computer service

When more detailed instructions must be given to the computer, a second function must usually be performed. This function is the translation of the language in which the instructions are given into machine language. In this process, the source program serves as input to the translator program, and the output is the program listing and the corresponding object program which can then be run on the machine. Figure 2.11 illustrates this process. The object program usually stays in the sys-

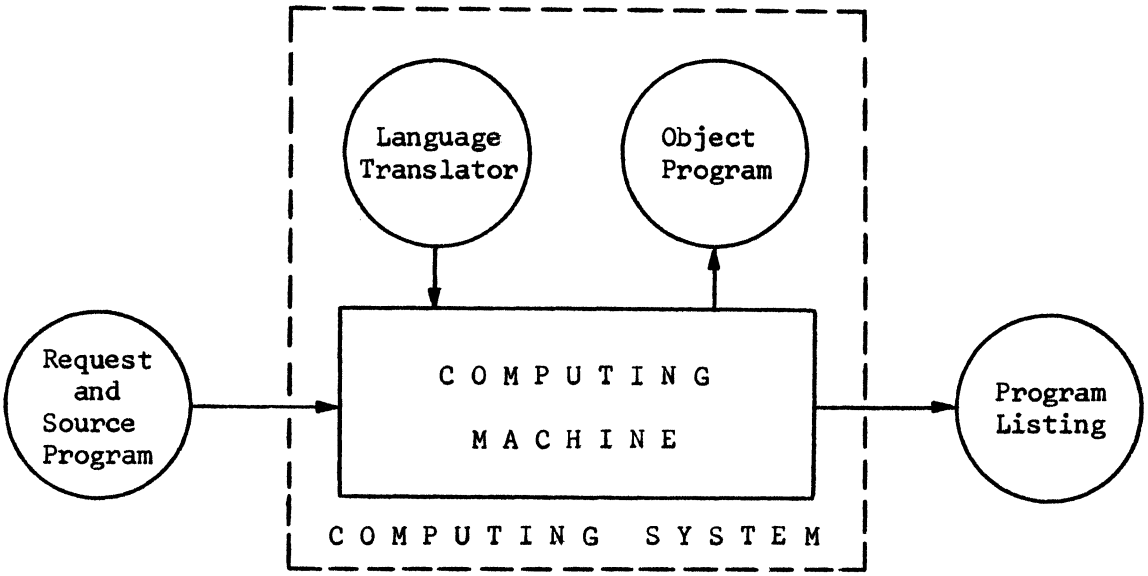



Figure 2.11 Translation of a source program

tem at least temporarily, and the user only gets a printed listing of his source program with comments by the translator as to any errors it may have detected. If he makes further requests he can also get a listing of the object program, a copy of the object program on tape or cards, and other information.

When the user wishes to apply a newly translated program to some data, a second request is made and the data provided or identified. In most present systems this second request is added to the first by simply asking for the system to execute the program. In batch processing with a procedure oriented source language such as MAD one asks for compilation with execution, furnishes the source language of the program, and follows this by the data. Figure 2.12 illustrates this mode of operation.

Before execution can take place the user's routines and any library routines which are used must be loaded into the memory of the machine. As the loader is putting these routines into memory it usually produces a map for the user showing precisely where each routine has been located. In the systems having the MAD compiler, this map designates the main routine with (MAIN) and is usually fairly long even for

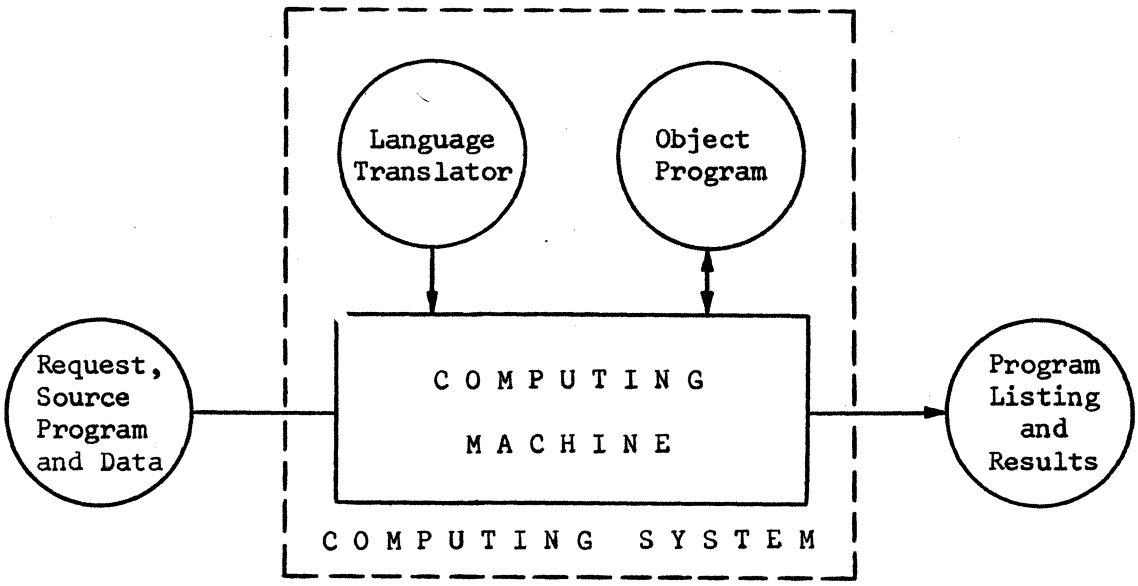



Figure 2.12 A combined translation and execution

simple programs. Many small library and system programs are used for input-output and various other purposes. Although this map is of little help to the inexperienced user, it becomes most valuable on occasions to those with more understanding of the machine and the system. If the memory requirements of a program exceed those available, then the loader usually terminates the job with an appropriate diagnostic.

During the execution of a program, other error messages may be printed. In systems using MAD, these are printed after the memory map and along with any output from the program. Thus it is easy to distinguish between diagnostics produced by the MAD compiler and ones produced at run time. Most of these latter diagnostics concern input-output, arithmetic operations, or subroutines. For example, they might state that an input card has illegal characters on it, that there has been an overflow during an attempt to add or multiply two numbers, or that some function is undefined for the argument given. As a particular example of this latter case, the MAD system will, on valid grounds, refuse to take the square root of a negative number and tell the user about it if his program attempts to do so (see Appendix C).

Thus one can see that the overall effectiveness of an organization which uses a computer system is quite dependent on the software available and the design of the system. With good software and system design the work of the user may be made more convenient and efficient. In fact, a well designed computing system can perform in a somewhat intelligent manner; however, the more advanced systems are all the more incomprehensible when they fail to act properly.

Chapter Three

# DESCRIPTION OF LANGUAGE AND DATA

Since we wish to develop some understanding of the structure of the languages that are used with computers as we describe and use the MAD language, a notation for precisely describing such languages is introduced in this chapter. After discussing the overall structure of MAD programs and routines, this notation is used to define the more simple types of data objects which are used in the MAD language. Finally, we present the basic statements in the MAD language which are used to cause the input of data and the output of results and comments.

## 3.1 The Description of Languages

Most written forms of Indo-European languages are constructed from a set of marks or characters which may be combined to form words. These words are then used to form expressions, phrases and sentences in accordance with certain rules. The collection of these rules which define part of the structure of the language is called the syntax of the language. There are also rules concerning the formation of words, paragraphs and more complex structures, but these are not usually included under the category of syntax. For example, in the English language, a simple sentence may have only two formal parts, a subject and a predicate, and these parts may vary widely in complexity. For example, the two sentences,

"People think." , and

"Both people and computers can do routine mental work."

have the same syntactic structure of the form subject predicate.

The meanings of words, as well as more complex structures, in a language are subject to agreement among those who use the language and are called the semantics of the language. Thus, the word "ring" may mean different things to a prize fighter, a jeweler, an automobile mechanic, a telephone operator, and a mathematician. Whole sentences, and even paragraphs, out of context may completely lose their meaning. Furthermore, it is possible to make syntactically correct constructions that are not likely to be useful in any context. For example, the English sentence,

"Digits think.",

falls into this category.

In developing procedure oriented languages for communication with machines, it is necessary to consider both syntax and semantics. In these languages it is customary to use the term syntax to cover the whole range of structural rules varying from those for numbers, words, statements, and declarations to those for whole routines and programs. Since the converters which take these languages as input and produce machine language use primarily the syntax of the language to resolve ambiguities, they are not as accommodating as people in understanding languages. Thus, it is necessary to define these languages more precisely and with more restrictions than natural languages. The user of these languages must adhere strictly to the syntax of the language, but, as with natural languages, this does not in itself guarantee that he will produce meaningful statements. The language translator for most mechanical languages will detect essentially all syntactic errors and either attempt to make corrections or simply reject programs containing such errors. In either case, a message is usually given to the user indicating the trouble. On the other hand, less can be done in the case of semantic errors. For example, the translator would be unable to determine that the user really meant the sum of A and B, although he wrote down the product.

In order to describe the syntax or structure of a mechanical language precisely, it is convenient to use an intermediate language or notation called a metalanguage. Such a metalanguage, which has been found to be especially valuable for describing languages similar to MAD, will now be introduced and used throughout the text. This notation is a modified form of the Backus Normal Form or Backus Naur Form (BNF) notation[1]. This metalanguage is not itself part of the MAD language. Furthermore, it cannot be used to describe the meaning of the MAD language. It is used only to describe the syntax of the MAD language. We will not apply this notation completely rigorously, but on the whole will give our descriptions using it.

In the metalanguage, various metaexpressions may be built up from two types of elements called metaconstants and metavariables. A metaconstant indicates the literal occurrence of the specific character or characters which appear. Some examples are: FØR, W'R, A, +, and $(0). A metavariable denotes a general class of syntactic structures which may occur in the language. These metavariables are written with English words, abbreviations and numbers possibly connected by hyphens and always enclosed in angular brackets. Examples of metavariables are: <letter>, <exp>,

[1] J. W. Backus, The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. Proc. Internat. Conf. Inf. Proc., UNESCO, Paris, June 1959.

<f-p-con-1>, and <array-structure>. A metaexpression consists of sequences of one or more metaconstants and/or metavariables or combinations of these sequences connected by certain operators to be discussed below. For example, A(<digit>) + 4 and AD or BC are metaexpressions. A metavariable will be defined by placing it to the left of the word is and putting its defining expression on the right. The operator or which will be used to combine alternate parts of the definition is used in the exclusive sense to mean that exactly one of the forms or structures combined is to be satisfied. In some cases verbal definitions are the most convenient and will be used. Spaces occurring in these defining metaexpressions are not significant.

The most simple form of this notation is used here to define the basic character set used in the MAD language.

Definition of letter, digit, symbol, blank and character:

| | | |
|---|---|---|
| <letter> | is | A or B or C or D or E or F or G or H or I or<br>J or K or L or M or N or Ø or P or Q or R or<br>S or T or U or V or W or X or Y or Z |
| <digit> | is | 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9 |
| <symbol> | is | + or - or * or / or , or . or = or ' or $ or ( or ) |
| <blank> | is | an empty space of the same size as that usually occupied by a letter, a digit or a symbol |
| <character> | is | <letter> or <digit> or <symbol> or <blank> |

In most places in the MAD language, blank spaces have no significance, although they are meaningful in character strings. Where it is necessary to indicate the occurence of a blank in a structure being defined, the symbol Δ will be used to represent it. The letter Ø will be slashed to distinguish it from the digit zero (0).

Since none of the above definitions have involved metaexpressions other than simple metaconstants and metavariables, they do not demonstrate the real power of this metalanguage. To illustrate the power of this notation, we define a small subset of the English language which might be used by children in reporting deportment on a school playground.

Definition of given name, tattle verb, and tattle tale:

| | | |
|---|---|---|
| <given-name> | is | JØHN or PHILIP or HENRY or<br>MARSHA or ANN or KATHY |
| <tattle-verb> | is | HIT or BIT or KICKED or<br>PINCHED or SCRATCHED |
| <tattle-tale> | is | <given-name> Δ <tattle-verb> Δ <given-name> .<br>or TEACHER, Δ <tattle-tale> |

Thus, examples of a tattle tale are:

HENRY HIT MARSHA.

TEACHER, JOHN KICKED KATHY.

KATHY SCRATCHED PHILIP.

TEACHER, TEACHER, ANN PINCHED JOHN.

TEACHER, ... , TEACHER, JOHN BIT MARSHA.

The above illustration points out very clearly that with the use of more complex metaexpressions in these definitions the metalanguage becomes a powerful tool. The fact that constructions of questionable value, although syntactically correct, are allowed is demonstrated by

TEACHER, ANN HIT ANN.

We will later introduce ways of eliminating some such meaningless constructions.

An important feature, called recursiveness, of this type of definition is shown in the above illustration. Because of the way the metavariable <tattle-tale> is used in its own definition, an arbitrary, but finite, number of occurrences of TEACHER,Δ may begin the tattle in an instance of the class of sentences defined. The recursiveness of these definitions is not necessarily contained in a single one of them, but may involve several such definitions which interact with each other.

Since the MAD language is basically recursive in nature, the value of this feature will become more apparent with use. Frequently our definitions must involve metavariables which cannot be defined until much later in the text and these will be marked with a number sign (#) to warn the reader that they have not yet been defined.

In some cases it is necessary to represent a character which is a letter or a digit, and this can be done by defining a new metavariable.

Definition of alphanumeric characters:

<alphanumeric> is <letter> or <digit>

Often such a new class name is used so infrequently that it is difficult to justify its introduction. To assist in making the definitions more readable and meaningful, the braces { and } , are used to enclose parts of a metaexpression in which alternate definitions are used to avoid assigning a metavariable. For example,

{<letter> or <digit>}

could be used to replace the above metavariable <alphanumeric> anywhere in a metastructure.

The definition of a "word" using the metalanguage could be given by:

<word> is <letter> or <word> <letter>

The set of such "words" would include all normal words in the English language, plus many others. Since this definition is recursive, it includes all "words" of

arbitrary, but finite, length. In some cases it is desirable to define "words" or structures having either a fixed length or a length within specified ranges. Such a "word" or structure is actually a string or sequence of letters or other structures, so we introduce a string operator S which may be written as follows:

$$S_2^5\{\text{<letter>}\}$$

The metaexpression to be repeated is enclosed in braces, and the string operator S with its bounds precedes these braces and indicates that the enclosed metaexpression may be repeated any number of times between and including the bounds. The above example then represents all "words" having two, three, four or five letters. More often the lower bound will be 0 or 1, but it is always stated. If the lower bound is 0, then the string defined may be empty. The empty or null syntactic structure is often useful and so its definition should be made specific.

Definition of the null structure:

<null> is

The upper bound is sometimes arbitrary and this case is indicated by using an asterisk for the upper bound. Thus

$$S_1^*\{\text{<letter>}\}$$

defines the same syntactic structure as is represented by <word> above. For strings of fixed length the two bounds are the same and so

$$S_4^4\{\text{<letter>}\}$$

represents the collection of all four letter "words", such as PFST.

In many instances in the MAD language, lists of various syntactic structures are allowed where the elements of these lists are separated by commas or other characters. Such a non-empty list of structures each denoted by <s> could be defined by:

<list-of-s> is <s> or <list-of-s>, <s>

Once again to make it unnecessary to introduce so many new metavariables, a list operator L is introduced which is written as

L{<s>}

and defines the same structure as is represented by <list-of-s> above. Since commas are most frequently used to separate lists, they are assumed unless another separator is indicated. The separator can be specified by putting it between the operator L and the left brace. Thus

L*{<letter> or <digit>}

defines a list of letters or digits separated by asterisks. For example, A*2*C, 3*X*4*B*C*2 and W have this structure. The metaexpression operated on by the list

operator is assumed to occur at least once unless otherwise indicated, but bounds may be used with them just as with strings.

In some definitions the same instance of a metavariable must be used more than once. It is therefore necessary to have some notation for an arbitrary but fixed instance of the class denoted by metavariable. Subscripts on the metavariable will be used for this purpose. Thus, we could use

$$\text{<letter>}_1 \ \text{<digit>} \ \text{<letter>}_1$$

to denote all those "sandwich" type structures of three characters having the same, but an arbitrary letter, at each end and some digit in between. For example, A1A, C7C, and P2P. This notation is also useful when it is desired to refer to one or more of the metavariables in the discussion.

It is sometimes desirable to use a metaexpression which represents a class of constructions very close to one which is already defined, and obtained from it by excluding some particular element or elements. An operator written as not is used for this purpose. As examples,

$$\{\text{<letter>} \ \underline{\text{not}} \ \text{Q}\}$$

represents the twenty-five letters other than Q and

$$\{\text{<character>} \ \underline{\text{not}} \ \text{<blank>}\}$$

represents a character other than the blank.

Finally, since blanks are ignored in most places in MAD, it will be assumed that any number of blanks may be inserted anywhere in a structure unless otherwise stated. Thus, A,BA B,B B, and BAB are all examples of

$$\underline{S}_1^3\{\text{A} \ \underline{\text{or}} \ \text{B}\}$$

Exercises

1. Use the metalanguage introduced in this section with <d> as an abbreviation for <digit> to describe the general class of amounts of money indicated (in dollars and cents with the leading digit of any dollar amount being non-zero):

a) Amounts from $ .00 to $ .99 inclusive.

b) Amounts from $ .00 to $9.99 inclusive.

c) Amounts from $ .00 to $99.99 inclusive.

d) Amounts from $1.00 to $99.99 inclusive.

e) Amounts from $ .00 to $10,000.00 inclusive.

f) Amounts from $ .00 up to but not including $2,000,000.00.

g) Arbitrary amounts.

2. What class of numbers is represented by the following metalanguage description?

$$\underline{S}_2^2\{\text{<letter> } \underline{\text{not}} \{Q \underline{\text{ or }} Z\}\} \quad \{\text{<digit> } \underline{\text{not}} \ 0\} \quad \underline{S}_4^4\{\text{<digit>}\}$$
$$\underline{\text{or}} \ \underline{S}_2^2 \ \{\text{<digit> } \underline{\text{not}} \ \{0 \underline{\text{ or }} 1\}\} \ \underline{S}_5^5 \ \{\text{<digit>}\}$$

3. The Area Codes for the Direct Distance Dialing system which has been set up by the telephone companies in the United States consist of three digits, the first of which cannot be 0 or 1, the second being either 0 or 1 and the third never being zero. Describe these area codes in the metalanguage.

4. Combine the description produced in problem 3 with a description of the most common type of local telephone numbers to obtain a general description of "telephone numbers" which would allow one to make both local and long distance calls.

5. Using the metalanguage introduced in this section, describe the classifications which may be assigned to books under the Dewey Decimal System.

6. Using the metalanguage introduced in this section, describe the classifications which may be assigned to books under the Library of Congress classification system.

## 3.2 Structure of MAD Programs

As stated in Chapter One, a computer program or more simply a program, is a complete set of instructions given in one or more languages for solving a problem on a computer or a group of computers. Almost all programs are broken down into sections which we shall call routines. One of these routines normally plays a central role in controlling the other routines and is therefore called the main routine. A MAD program is one whose main routine is written in the MAD language. The other routines of a program which are controlled by the main routine are called subroutines. Many such subroutines are built into most modern computer programming systems. Some of these, called system subroutines, are automatically supplied by translators, loaders, and other system programs to handle certain processes such as input-output. Others, called library routines, are provided whenever another routine explicitly calls for their use or may be supplied by translators when needed. Every MAD program uses one or more system subroutines, and library subroutines are very frequently required by such programs. In addition to subroutines which are built into the system or library, the user may supply subroutines which are required by a main routine or his subroutines. These subroutines may also be written in MAD, but

there are several other ways in which they may be presented. For example, they may be written in assembly language which must be assembled by another system program, or they may have already been compiled or assembled and the object code stored in a compact binary form on cards or tape. It is also possible in MAD and some other programming systems to include subroutines as part of another routine, in which case, they are often called internal subroutines. In addition, a MAD subroutine may be used by a main routine written in an assembly or some other language. In this text we will program most of our algorithms as MAD main routines and MAD subroutines. Thus, these other topics will not be discussed further.

A MAD routine is made up of statements which may be of three types. An executable statement specifies some process or action which is to be taken by the computer. Such a statement causes the compiler to generate machine language instructions to carry out the operations involved in the process specified. Usually several machine instructions are generated as a result of a single MAD executable statement. When these instructions are obeyed by the machine we say that the MAD statement is executed. Statements of the second type, called declarations, provide information to the compiler itself, but do not cause any action during the execution of the routine and do not cause the generation of machine instructions. On the other hand, declarations do cause memory space to be reserved, give information about the mode and format of operands, and generate certain structural information used by the routine. In some other computer languages, declarations may actually cause processes to be performed when a routine is executed. The third type of statements, called remarks, is provided for in the MAD language only for the benefit of the people who are reading the statements. These remarks may be inserted to give a routine a title or to explain how the routine accomplishes something. Once again, remarks do not cause the routine to produce any action by the machine, they are not used by the compiler, and they are included as part of the printed listing of the routine only for the information of the user.

The statements of a MAD routine may be typed or written in many ways, but usually they must be punched into cards in order to present them to the computing machine. Certain fairly rigid rules must be observed in preparing these cards. Column 11 of the card may be used only for the special purposes of indicating remarks or continuation of other statements. Columns 73 through 80 of the cards are reserved to identify the cards as the user may desire and are not used by the compiler in any way other than to print them on the output listing.

Since the main part of all three types of MAD statements is punched in columns 12 through 72 of the cards, each card may have at most 61 characters of a statement

on it. The first card of such a statement must have a blank in column 11 and if the statement is to be continued on more than the one card, each additional card must have a digit punched in column 11. Any digits may be used in any order and as many such continuation cards as desired may be used. It is often convenient to number the continuation cards in order with 1, 2, 3, etc. There is, however, a limitation of 627 significant characters per statement where blanks in character strings between $'s and the full expansion of MAD abbreviations[2] must be counted. In general, blanks can be inserted in statements to make them more readable and the use of this freedom for indentation and other physical arrangement of statements can add greatly to the ease of reading the program.

A card which has an R in column 11 is treated as a remark. The R is not printed in the output listing and the contents of columns 12 through 72 of the card are often displaced one space to the left of the rest of the cards on this listing. Remarks may occur anywhere in a routine including between the cards of other statements which consist of more than one card. If a remark is to extend over more than one card, each of the cards used must have an R in column 11. A card which is completely blank in columns 1 through 72 is treated as a remark card.

Executable statements may contain a label in columns 1 through 10 of the first card of the statement. This label may be used to refer to the statement elsewhere in the routine, and if a statement is to be referred to elsewhere it must be so labeled. These labels are discussed in detail in Chapter 5. Punching in columns 1 through 10 of declarations, continuation cards and remark cards are ignored except for being printed in the output listing.

A special statement, which must be the last statement of every MAD main routine, is the END ØF PRØGRAM statement. This statement consists of

END ØF PRØGRAM

followed by any characters desired. This statement serves two purposes. First, it is a signal to the compiler that there are no further statements to be translated in that routine. Second, if executed, this statement causes a return to the system and termination of the program. It may not appear elsewhere in a MAD routine.

The remaining statements will be introduced as they are needed and, in particular, the structure of subroutines will be discussed in Chapter Seven.

---

[2] Many key words and phrases may be abbreviated in MAD. A list of these abbreviations is given in Appendix A.

## 3.3 Identifiers and Mode Declaration

Many parts of the MAD language must be identified or named, including variables, subroutines and some statements. It is thus natural that the definition of an identifier, which will be used for that purpose, be given as one of the first syntactic entities.

Definition of an identifier:

<id> is <letter> $S_0^5${<letter> or <digit>}

Some examples of identifiers are L4, START, Q493D, SAM, SQRT, and PAYDAY. The following are not identifiers for the reason indicated:

| | |
|---|---|
| ABCDEFG | More than six characters. |
| 3Y | Begins with a digit. |
| π | Not a MAD character. |
| CD$Q | $ is not a letter or digit. |
| A(3) | ( and ) are not letters or digits. |
| C1.3 | . is not a letter or digit. |

The processing which is describable in the MAD language involves operations on elements of various modes, usually integer, floating point or Boolean. Each of these elements occupies a single word of memory or storage and is operated on as a unit. An individual element which is to remain fixed throughout the execution of a routine may be treated as a constant and these will be discussed in section 3.4. If an element is to have its value changed during the execution of a program, it must be identified or given a name so that reference can be made to it. Elements of this type are referred to as variables. Often it is convenient to use a structural collection or array of related variables for some common purpose and to give them a single name. In this case, it is customary to refer to the individual elements of the collection by means of subscripts applied to the common name. One or more such subscripts may be used, but we will discuss mainly single subscripts in this chapter. Higher order arrays and multiple subscripting will be discussed in Chapter 8. In all arrays, however, the mode or type must be the same for all the elements involved.

In working with variables and arrays, three factors must be specified to the compiler, namely their mode, how much memory they are to occupy, and their structure, if any. From this information the compiler can arrange for the proper space and structural information to be made available and associated with the identifier of the variable or array and to provide for the conversion of variables between

modes when necessary. In the case of individual variables the only thing which must be specified is the mode since they occupy a single memory word and have essentially no structure. In this section we will discuss only the mode of variables and arrays leaving the structure of arrays to section 3.6 and Chapter 8.

In addition to the modes used for variables (integer, floating point, and Boolean) there are two other standard modes which are used in the MAD language. One of the other standard modes is statement label which means that the identifier or name is used as a label on a statement. The other standard mode is function name which is used to classify identifiers which are used as the names of functions, procedures, or subroutines.

Definition of a mode name:

<mode-name> is INTEGER or FLØATING PØINT or BØØLEAN
or STATEMENT LABEL or FUNCTIØN NAME

The mode of all identifiers is assumed to be the normal mode unless they are automatically declared by the compiler or unless a specific declaration to the contrary is made for individual identifiers. The normal mode is floating point unless declared otherwise. The normal mode of a routine may be changed by use of the following declaration:

NØRMAL MØDE IS <mode-name>

Only one such declaration may occur in a routine and it is effective throughout that routine. For example if a routine is to involve mainly integer mode variables and arrays the declaration

NØRMAL MØDE IS INTEGER

would save making so many individual declarations.

Automatic assignment of the mode of identifiers is made in the following cases:

a) An identifier used as a label on an executable statement is assigned statement label mode.

b) An identifier used as a function name is assigned function name mode.

c) A vector which is preset by use of the vector values declaration is assigned the mode of the first constant appearing in the declaration.

d) A vector appearing as the dimension vector in the structure of an array is assumed to have integer mode.

The mode in which an identifier is to be used in a MAD routine can be declared explicitly by the user in a declaration of the form:

<mode-name> L{<id> or {<id> or (L{<id>})} (<structure>#)}

#To be defined later in text. See section 3.1 for explanation.

Simple examples of such declarations are:

```
INTEGER I, J, KARØ
BØØLEAN B, AB, B4
STATEMENT LABEL START, L14
```

More complex examples which also specify the extent and structure of the operands are:

```
INTEGER A(10), (C,D,E)(10*10)
BØØLEAN B(100,BVD)
```

The specification of extent and structure will be discussed in detail in section 3.6 and Chapter 8. Although there is no restriction on the number or location of these mode declarations, a particular identifier must be declared to have only a single mode including the automatic assignment of mode. Note that if normal mode is declared to be integer, any floating point variables or arrays could be declared in a statement such as

```
FLØATING PØINT X,Y
```

which would not be needed when the normal mode was floating point. Also, redundant mode declarations are allowed, e.g. the mode of an operand may be declared more than once, as long as these declarations do not conflict.

In various places in the MAD language and in diagnostic statements, the standard modes are represented by a single digit as shown in Table 3.1. In addition to the standard modes discussed in this section, there are additional modes, represented by the digits 5, 6, and 7, which are available to indicate modes of variables

Table 3.1 Mode digits

| Digit | Mode |
|---|---|
| 0 | floating point |
| 1 | integer |
| 2 | Boolean |
| 3 | function name |
| 4 | statement label |

and arrays for special operations which may be defined in MAD. Further information on these modes may be found in the appendix on the DEFINE Facility of the MAD language. Since these facilities are infrequently used, they may be ignored by the beginning user.

Exercises

1. Which of the following are legitimate MAD identifiers? If not, why not?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a) | 23AB | d) | -ABC | g) | AB3.7 | j) | E2 |
| b) | A6432 | e) | ACCENT | h) | AB( | | |
| c) | A(6432) | f) | MARCØVITZ | i) | 3E5 | | |

2. If the following mode declaration is used, how are each of the calculated variables stored?

```
INTEGER  A, B, C
```

a) B = -5

b) A = -2

c) C = 3.694

d) E(4) = 4.9

e) A(2) = 3

f) X(7) = 4.5

g) X(8) = 5

3. Which of the following are legal MAD declarations? Explain what is wrong with those that are not.

a) BØØLEAN  X1, Y2

b) INTERGER  A, B, C

c) NØRMAL MØDE IS BØØLEAN

d) FLØATING PØINT MØDE

e) INTEGER IS NØRMAL MØDE

f) BØØLEAN A, B(1), BØØLEAN

g) NØRMAL MØDE IS 3

h) INTEGER I, J, K1, SAMUEL

4. What is the difference between X1 and X(1)?

## 3.4 Constants

It is often convenient to write numbers and other elements explicitly in programs where they are to be used in computations or manipulations. For example, the 1, 2, and 3.14159265 in the expressions I + 1 and 3.14159265*R.P.2 are such explicit numbers. Such elements which are written explicitly rather than being referred to by name, are called constants. Actually there are several classes of constants used in MAD programs. The operand constants which may be used in such operations as addition and multiplication form one such class. Constants of another type, called compilation constants, are used at the time of compilation to indicate the structure of operands, tape numbers and the like. Finally, the input data are a collection of constants to which the program is applied in a particular instance.

When an operand constant appears in a MAD routine, it is converted to machine representation by the compiler and stored in a word of memory. Thus, each constant occupies a single word of memory and the address of this word is used to refer to the constant in combining it with other elements. If several such constants having the same absolute value appear in a single routine, only one copy of the absolute value is stored and the same location is used in each of the appropriate places. Thus, although one may think of these constants as having signs, these signs are actually used as the unary operators + and - which will be discussed in detail in Chapter 4. An operand constant is also assigned a mode by the compiler depending on certain conventions or the user's explicit indication of the mode as discussed below.

Since in some instances it is desirable to think of operand constants as having a sign attached and furthermore signed constants may be used in input data, it is desirable to make clear what is meant by a sign.

Definition of a sign:

<sign> is + or <null> or -

where the null sign is equivalent to the plus, but the minus must be written if it is meant.

There are six ways of writing unsigned constants available in MAD, namely, decimal (integer), octal, character, floating point (two forms) and truth constants. These will now be discussed individually.

The most natural way of writing integer constants is as decimal integers. As is customary in mathematics, integers are not written with a decimal point and, in fact, in MAD a decimal point must not be used since it would signal a floating point number. There is, however, an assumed decimal point to the right of the last

digit.

Definition of an unsigned decimal integer constant:

<deci-con> is <digit> or <deci-con> <digit>

These could be defined equivalently by:

<deci-con> is $S_1^*\{<digit>\}$

Although leading zeros have no effect on the value of a decimal integer constant, trailing zeros each indicate a factor of ten in the size of the integer. Because of the finite length of machine words, there is a limitation on the size of constants that may be used. In MAD, this limit is 34359738367, i.e. $2^{35} - 1$, because of the 36 bit word length of the machines on which MAD is implemented. If a larger integer constant is used, a diagnostic message is printed as follows:

***** ERROR 35455 .... CONSTANT OUT OF RANGE.

If the integer constant is to be converted to floating point representation, then it must be less than $2^{27}$ or 134217728 in magnitude for the result to be meaningful. Decimal integer constants are normally assigned integer mode. Examples of this class of constants are: 7, 463, 29, 0, 4692, 46, 046, 00046 and 000000046 where the last four of these are equivalent.

Octal constants are written as a string of one to twelve octal digits followed by the letter K and possibly a decimal (integer) constant less than 12.

Definition of octal digit and octal constant:

<octit> is 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7

<octa-con> is $S_1^{12}\{<octit>\}$ K or $S_1^n\{<octit>\}$ K<deci-con>

where n must be less than twelve and the decimal integer lies between 0 and 12-n. The unsigned decimal integer constant following the K indicates powers of eight (i.e. the number of trailing zeros to be added). Thus, 20450000000K and 2045K7 are equivalent, each producing the octal machine word 020450000000. Octal constants are normally assigned integer mode.

Character constants provide a way to write elements which represent one or more of the 48 characters available in MAD. These constants are enclosed by dollar signs as brackets.

Definition of a character constant:

<char-con> is \$ $S_0^6\{\{<character>$ not \$\} or \$\$\}\$

Since a second $ in the string could mean the end of the string of characters, two adjacent $'s are used to represent a single $ in the string. Between the $'s used as brackets in these constants, blanks are significant. If the constant consists of less than six characters, blanks are added to the right to make six characters.

If the string is empty, it is treated as six blanks. The six characters of these constants are stored using six bits for each character in accordance with the octal storage code shown in Table 2.3. Character constants are normally assigned integer mode. Examples of valid character constants are: $24D$, $MADAMI$, $1...99$, $C=-A+B$, $$, $31$, $ $$3.95$. Both $ $9.98$ and $0123456$ are invalid, the first having an odd number of $'s and the second having 7 characters between the $'s.

Floating point constants may be written in two forms and must be used to represent numbers which may have fractional parts and those outside the legal range for integers. In the first form, the occurrence of a decimal point indicates a floating point number such as 7. as opposed to the integer 7 appearing above. The difference in this case is in the way the number is stored and used in the routine. Examples of this form of floating point constant are: 7123.45, 3.14159265, 0.0000, .2 and .00001694. Formally we give the

Definition of floating point constant (first form):

<f-p-con-1> is <deci-con>. or .<deci-con> or <deci-con> . <deci-con>

The second form of floating point constant involves the use of an exponent of the power of 10 which is multiplied by an integer constant or a floating point constant of the first form. Thus formally we give the

Definition of floating point constant (second form):

<f-p-con-2> is {<deci-con> or <f-p-con-1>}E<sign>$S_1^2${<digit>}

Examples of this second form are: .46E-9, 2.9743246E10, 1E10, 493.72E-5, .0000001E10 and 7.E3.

As discussed in Chapter 1, these floating point constants are represented in the machine using a mantissa and a characteristic in a manner similar to scientific notation. Because of the limited word length of the machines only a small number of digits can be represented. Although as many digits may be written in a MAD routine as desired, only about the first eight significant decimal digits of the mantissa will be used. There are also limitations on the effective characteristic to an approximate range of -38 to +38. To be precise, the value of a floating point constant f must satisfy the relation

$$.1000000 \times 10^{-37} \leq |f| \leq .17014112 \times 10^{39}$$

or be zero. As might be expected, both forms of these floating point constants are normally assigned floating point mode.

Truth constants may take on only two values representing "true" and "false".

Definition of a truth constant:

<truth-con> is 1B or 0B

where 1B stands for "true" and 0B for "false." These truth constants are normally assigned Boolean mode.

The general class of unsigned constants may now be defined in terms of these six ways of writing them.

Definition of an unsigned constant:

<uns-con> is <deci-con> or <octa-con> or <char-con>
or <f-p-con-1> or <f-p-con-2>
or <truth-con>

When the normally assigned mode of a constant is not desired, the mode may be explicitly indicated by a suffix called a mode indicator.

Definition of a mode indicator:

<mode-ind> is M <octit>

where the octal digit has the meaning discussed in section 3.3. Although this type of mode indication is not used frequently in constants, it is included in the definitions for completeness.

We are now able to give the general definition of an unsigned integer constant. These constants are required in certain places in the MAD language especially as compilation constants.

Definition of an unsigned integer constant:

<uns-int-con> is <deci-con> or <octa-con>
or <char-con> or
<uns-con> M1

It must be emphasized here that these mode indicators must be used with understanding and care. They do not change what is stored in memory, but only the mode in which the constant is treated. For example, 7E2 M1 will cause a floating point 700.0 to be stored in memory, but this number will be treated as an integer--a very large integer.

Using the definition of a sign, we now define an integer mode constant.

Definition of a signed integer constant:

<int-con> is <sign> <uns-int-con>

Some examples of signed integer constants are: 0, +0, +46, 046, -46, -169432, and +000231.

The general definition of signed and unsigned floating point constants can now be given by combining the two forms of floating point constants introduced above and using the special form of floating point constant where the mode is explicitly

indicated.

Definition of unsigned and signed floating point constants:

<uns-f-p-con> is <f-p-con-1> or <f-p-con-2>
or <uns-con> M0

<f-p-con> is <sign> <uns-f-p-con>

The writing of a floating point constant as an octal constant with the explicit mode indicator is sometimes useful when the exact bit structure of the constant to be specified is important. For example, 2034K8M0 is a precise 4.0 and 1764K8M0 is 1/8.

Since the combined group of integer and floating point constants may be used in arithmetic operations, it is convenient to give a definition of a metavariable which may be used in later definitions.

Definition of an arithmetic constant:

<a-con> is <int-con> or <f-p-con>

Another class of constants which are available in the MAD language are the Boolean constants which will be used in Chapter 5.

Definition of a Boolean constant:

<bool-con> is <truth-con> or <uns-con> M2

Thus a Boolean constant may be either 1B or 0B, standing for "true" and "false" respectively, or any constant with an explicit mode indicator of Boolean mode. The explicitly indicated mode allows the "true" Boolean constant to be written as 1M2 if desired and other similar, but not necessarily useful, things to be written.

Finally, we can now define the most general class of constants in the MAD language.

Definition of a constant:

<constant> is <a-con> or <bool-con>

Most of the classes of constants which have been defined in this section will be used in definitions later in the text to make clear just what constructions are legal in various circumstances. Further reference to this section as these constants are used will probably be necessary.

Subroutine constants, i.e. constants of function name mode, are quite different in structure and use from the constants described in this section and so they will be discussed in Section 7.1 in conjunction with subroutine calls.

Exercises

1. Express the following MAD constants in normal decimal notation indicating which are in integer mode.

a) 07100

b) 07100.0

c) 71E3

d) 71.E3

e) 71000E0

f) 71.0E4

g) 71E-1

h) -07.10E-6

2. Write a correct expression for each of the following numbers in both MAD floating point forms, and in integer form when possible.

a) 263

b) $31.2 \times 10^{-3}$

c) $18.416 \times 10^{-4}$

d) .0314962

e) $14 \times 10^{6}$

f) 10000

g) $-31 \times 10^{-2}$

h) $.00162 \times 10^{8}$

i) $.0032 \times 10^{39}$

3. For each of the numbers on the left write the letter or letters of all MAD constants on the right which represent that number.

a) 10.0

b) $10^{-2}$

c) -1.

d) $0.1 \times 10^{2}$

A) 1E1
B) 10.0E-1
C) 0.01
D) -1.000
E) 100000E-4
F) 100.000
G) -1E0
H) 1.0E-1
I) .0000001E+5
J) -10.0E-2

## 3.5 Variables and Subscripts

When an element may have its value changed during the execution of a routine, it is called a variable. If this variable corresponds to an individual element then it has its own identifier or name and is referred to by that name. If the variable refers to part of an array of elements having a common name, then it is referred to by using subscripts on that name.

Definition of a variable name:

<var-name> is <id>

where the name and thus, the corresponding element or elements having that name, are declared to be of integer, floating point, Boolean, or one of the special modes. A variable itself is then a reference to a single element, individual or part of an array.

Definition of a variable:

<var> is <var-name> or <var-name> (<subscript-list>)

Although the precise meaning of a subscript list will not be completed until the next chapter, it is given here followed by a verbal explanation and numerous examples of subscripted variable names.

Definition of a subscript list:

<subscript-list> is <a-exp>$^{\#}$ or <a-exp>$^{\#}$, <a-exp>$^{\#}$

or $L_3^*$\{ <int-exp>$^{\#}$ \}

Verbally this definition may be explained as follows: In the case of either single or double subscripts, the subscripts may be arbitrary arithmetic expressions of either integer or floating point mode. Those which have floating point mode will be truncated to integer form when they are applied as subscripts. On the other hand, if more than two subscripts are used on a variable name, all of these subscripts must be expressions which have integer mode. The compiler will not convert floating point mode expressions in these cases, and, in fact, will not detect that they have been used. Examples of such variables in the MAD language are SAM, X(2), N(N), A(4.36*SIN.(X)+PI), W(E,4), A(N+3), T(T+5, 9.63*W), M(3*N, 46, 9, 0, 11) where N is integer and HUP(2,3,4).

The fact that a subscripted variable is syntactically correct does not imply that it represents a proper element of an array. In fact, the use of subscripts outside the range reserved in the program will not be detected by the compiler and if they are used in execution may have dire consequences on the user's program as well

as the computer systems programs. Close attention should be paid to the use of subscripts to insure that they will produce valid elements of the array. There will be further discussion of subscripts in Chapter 8.

We must often refer to variables whose values are of integer, floating point, or Boolean mode and sometimes to those which may be either of integer or floating point mode.

Definition of integer, floating point, arithmetic, and Boolean variable:

| | | |
|---|---|---|
| <int-var> | is | a variable which has integer mode |
| <f-p-var> | is | a variable which has floating point mode |
| <a-var> | is | a variable which has integer or floating point mode |
| <bool-var> | is | a variable which has Boolean mode |

It is the responsibility of the user to explicitly give each variable a value before it is used to compute other values and there are several ways of assigning these initial values. First, this may be done by an assignment statement such as

```
I = 1
```

which requires execution time, but has the value that it is reassigned each time the statement is executed. Second, the variable may be given a value by reading data into it. Third, a vector or array variable may be set to some value at compile time by the VECTØR VALUES declaration which will be discussed in section 3.6.

Although it should never be relied upon, the system does assign an initial value to variables when the user neglects to do so. The value so assigned is called the core constant and has been selected so that its use will most likely cause a detectable error. Table 3.2 shows some typical "values" which might appear on printed

Table 3.2 The core constant

| Value | Form |
|---|---|
| +007400403771 | octal word |
| 1006766073 | decimal integer |
| 0.00000 | fixed point decimal number |
| 1.884522E-37 | floating point decimal number |
| 0(0- Z | BCD word |
| TSX /Ø/3771,4 | assembly language instruction |

listings as a result of using this core constant. These values may vary slightly with the operating system being used. The appearance of such values in a printed listing is a potential signal of trouble and should be justified.

Since the core constant is very small as a floating point number, it often causes the results of computations to fall outside the range of the machine (called underflow or overflow). Its large value as an integer also frequently causes a failure in computations. If obeyed as an instruction, the core constant causes a transfer to the system which results in the following message:

```
**** WHAT WERE YØU DØING
AT LØCATIØN 10473
```

and termination of your program by the system. Thus, the core constant helps protect the system from the user and helps protect the user from his own errors.

Often it is convenient to assign a value to a variable which is then allowed to remain constant throughout a routine and use that variable in place of the constant. For example, one might let

```
PI = 3.14159265
```

early in the execution of the routine and thus be able to write only two characters rather than ten each time PI is used. Except for the time used in this initial assignment statement, this is not less efficient than the use of the constant in each instance where it is needed, but one must make sure that the statement giving PI a value is executed before PI is ever used.

## Exercises

1. Which of the following are not valid MAD variable names and why not?
   a) X2D
   b) T(3)
   c) 24D
   d) CHART124
   e) ATØZ
   f) J2.3
2. Which of the following cannot be valid MAD variables and why not?
   a) ABC
   b) X27.3
   c) SIN.
   d) N(23)
   e) K(N+1)
   f) H(M,S)
   g) R(2.7,5.3)
   h) M(2,1,3.6)
   i) T(1.3,2.1,5,7)
   j) C$Q(3)
   k) R(2*(A+B))
   l) A(B(C(D)))
   m) X(X)

## 3.6 Vectors and Initialization

As indicated earlier in the text, it is often convenient to have a number of elements combined into a structured array which is given a single name, where the individual elements are then referred to by applying subscripts to this name. In this section, we will treat the most simple case of this type, which is called a vector and which allows only one subscript. The structure of a vector is quite simple; elements with consecutive subscripts are stored in adjacent memory locations. One of these elements is called the base element of the vector and is used to locate the others by means of the subscript. In MAD, the base element is the one having the largest memory location and this element corresponds to the zero subscript. For example, the vector V having five elements might be stored in the machine as follows:

| | | | | |
|---|---|---|---|---|
| V(0) | in | memory | location | 12146 |
| V(1) | " | " | " | 12145 |
| V(2) | " | " | " | 12144 |
| V(3) | " | " | " | 12143 |
| V(4) | " | " | " | 12142 |

where all addresses are given in octal. Clearly the subscript may simply be subtracted from the address of the base element to obtain the address of any given element in the vector. The structure of all vectors is the same and so only their mode and extent must be specified to the compiler. The extent, which is given in terms of the largest subscript which may be used, can be included as structure in a mode declaration, can be listed in a separate DIMENSIØN statement, or may be implicitly given as a result of initializing the values of the vector in a VECTØR VALUES declaration. All of these will be discussed in this section.

The separate dimension declaration is of the form:

DIMENSIØN L{{<id> or (L{<id>})} (<structure>)}

Here the identifier (or each of the identifiers enclosed in parentheses) before a particular structure specification is given that structure, which in the case of vectors consists simply of an unsigned integer indicating the largest subscript which may be used. Since the more complex structures of higher dimensional arrays will be deferred to Chapter Eight, we will give only an incomplete definition of the metavariable <structure> at this time.

Definition of structure:

<structure> is <uns-int-con> or <array-structure>#

Since the base element has a zero subscript, the number of memory locations which are reserved by declarations of vectors is actually one greater than the largest subscript which may be used. As a simple example, if a routine contained only the declarations

```
DIMENSIØN   A(10), S(50)
INTEGER     I, S, N(21)
```

(and thus normal mode were floating point) then I would be an individual integer mode variable, S a vector of 51 integer mode variables S(0) to S(50), A a vector of 11 floating point mode variables A(0) to A(10) and N a vector of 22 integer mode variables N(0) to N(21).

The name of a vector in MAD is actually associated with the address of the base element, and so this base element may also be referred to by the name of the vector alone just as if it were an individual variable. Thus A, S and N refer to the same elements as A(0), S(0) and N(0) respectively; however, the unsubscripted version is obtained from memory more efficiently. Since in MAD the number of locations reserved for a vector is fixed at compile time the structure must be an unsigned integer constant, not a variable or an expression. Often the actual number of elements of a vector which are in use may vary depending on the problem being solved. In this case, the number of elements in the vector may be stored as the value of an integer variable. For example, one might read in as data

```
N = 5, X(1) = 1.2, 2.7, 3.4, -1.5, 7.6*
```

where N represents the number of elements of X which are specified. When the vector is in integer mode, it is often convenient to use the base element to represent the largest subscript actually in use at a particular time. Thus the vector H might be declared by

```
INTEGER   H(100)
```

but only seven elements plus the count read into it from the data card

```
H = 7, 3, 1, 6, 9, 4, 17, 5 *
```

The variable H, i.e. H(0), could then be used to indicate the number of elements of the vector containing data for this case. In particular, H(H) would be a proper designation for the last element containing data.

Some more complex examples of vector declarations are as follows:

```
DIMENSIØN   (A,AX,TØTAL)(74), T(25), B(0), MM(5)
BØØLEAN     (BY,BZ)(10), E, H(5), T
INTEGER     I,J,K(23), (M,N)(100), A(99)
```

If these are the only declarations in a routine, the mode and number of storage locations reserved for each variable and vector are shown in Table 3.3. Note that A and

Table 3.3 Results of example declarations

| Name | Mode | Extent | Vector |
|---|---|---|---|
| A | Integer | 100 | Yes |
| AX | Floating Point | 75 | Yes |
| B | Floating Point | 1 | Yes |
| BY | Boolean | 11 | Yes |
| BZ | Boolean | 11 | Yes |
| E | Boolean | 1 | No |
| H | Boolean | 6 | Yes |
| I | Integer | 1 | No |
| J | Integer | 1 | No |
| K | Integer | 24 | Yes |
| M | Integer | 101 | Yes |
| MM | Floating Point | 6 | Yes |
| N | Integer | 101 | Yes |
| T | Boolean | 26 | Yes |
| TØTAL | Floating Point | 75 | Yes |

T appear in two declarations. This is allowed as long as two different modes are not declared for the same name. The largest extent so declared is always used, and thus 100 spaces are reserved for A. Also note that the vector B has only one element but this element may be designated by B(0) whereas this subscript would be illegal if B were not dimensioned.

An identifier which is declared to stand for a vector or an array is required in certain positions in the MAD language.

Definition of a vector name:

<vector-name> is <id>

where the identifier has been declared to represent a vector or an array.

In a number of situations in the MAD language an arbitrary, but fixed, element of a vector may be used as the base of a subvector. Thus, it is useful to define a metavariable to represent such a base.

Definition of a vector base:

<vector-base> is <vector-name> or
<vector-name>(<uns-int-con>)

In other circumstances a fixed block of elements from a vector or an array which are specified by the end points of the block in terms of their linear subscripts is required.

Definition of a vector block:

<vector-block> is <vector-name>$_1$ (<uns-int-con>$_1$) ...
<vector-name>$_1$ (<uns-int-con>$_2$)
or <vector-name>$_1$ ... <vector-name>$_1$ (<uns-int-con>)

where <uns-int-con>$_1$ $\leq$ <uns-int-con>$_2$

Often it is useful to set the values of some or all of the elements of a vector at the time the routine is compiled. There are two forms of the statement which accomplishes this setting and the first form of this VECTØR VALUES declaration follows:

VECTØR VALUES <vector-base> = L{<constant> or <char-string>}

Strings of more than six characters are allowed in these declarations as a convenience in presetting messages, format descriptions and other character data.

Definition of character string:

<char-string> is $ $S_7^*${ {<character> not $} or $$} $

These character strings are broken up by the compiler into groups of six characters each and treated just as though they had been written as separate character constants. The last group is filled out with blanks on the right if necessary. As in character constants, blanks are significant in the character strings between the $'s.

The above VECTØR VALUES declaration causes the following things to be done by the compiler:

a) The base element of the subvector is given the value of the first constant in the list and successive elements of that vector or array are given the values of the remaining constants, in order, as they occur in the list. Other elements of that vector are not affected.

b) The mode of the vector is set equal to the mode of the first constant on the list.

c) Memory space is allocated for the vector to accommodate the largest subscript used in this process.

The constants in the list should all be of the same mode except in very unusual circumstances. Each constant is stored in the mode in which it is written. Since the vector is given the mode of the first constant in the list, any variation in

mode among the constants will usually cause those having mode other than that of the first to be treated improperly.

Some examples of the first form of the VECTØR VALUES declaration followed by their implicit mode and dimensioning consequences are:

```
VECTØR VALUES  DV = 2, 1, 8
     INTEGER  DV (2)
VECTØR VALUES  A(3) = 1.6, 2.9, 5.3, 8.1, -1.9, 2.6
     FLØATING PØINT  A(8)
VECTØR VALUES  B(0) = 1B, 0B, 0B, 1B, 1B
     BØØLEAN  B(4)
VECTØR VALUES  PI = 3.1415927
     FLØATING PØINT  PI(0)
```

This mode and dimensioning declaration is taken care of automatically in each instance by the VECTØR VALUES declaration and the second (indented) declaration in each case is not needed (but won't do any harm). Elements of only one vector may be set with a VECTØR VALUES declaration and these elements must be in consecutive locations. A maximum of 200 elements may be preset in a VECTØR VALUES declaration, but additional declarations may be employed to set further elements of the same array. The extent of the vector is automatically expanded as necessary; however, the first elements of such succeeding declarations must agree in mode with the previous mode which has been established. These declarations may occur anywhere in the MAD routine and succeeding declarations setting the same elements of a vector replace previous values assigned to those elements. Also, a VECTØR VALUES declaration may be used to set only a single element, but this always causes that element to be part of a vector, even if only a single element.

Elements of a vector which are declared but not set by a VECTØR VALUES declaration are assigned as their initial values the "core constant" as discussed in section 3.5. For example, the two declarations

```
VECTØR VALUES    X(2) = 2.7,3.9,-4.6
VECTØR VALUES    X(6) = 1.5,-2.6
```

would cause the vector X to have values as follows:

```
X(0)  =  core constant
X(1)  =  core constant
X(2)  =  2.7
X(3)  =  3.9
X(4)  =  -4.6
X(5)  =  core constant
X(6)  =  1.5
X(7)  =  -2.6
```

The second form of the VECTØR VALUES declaration is as follows:

VECTØR VALUES <vector-block> = <constant>

This declaration causes each of the elements in the specified block of the vector to be set by the compiler to the value of the constant. The mode of the vector name is set to that of the constant and memory space is allocated to accommodate the largest subscript in the block

An example of this declaration is

VECTØR VALUES X(7) ... X(21) = 2.713

which would set the mode of the identifier X to floating point, reserve 22 spaces for X, and preset each of the elements X(7), X(8), ..., X(20), X(21) to the value 2.713. As further examples the declarations

VECTØR VALUES A ... A(99) = 0

VECTØR VALUES A(0) ... A(99) = 0.

each store 0 in the first 100 locations of A and reserve 100 spaces in memory for A, but that the first declares A to be integer mode and the second declares it to be floating point mode.

Exercises

1. A program has to be written to find the sum of N numbers, A(0), ..., A(N-1) Depending on the circumstances, N may be 7, 10 or 15. It may never be anything else. Which of the following DIMENSIØN statements would be correct? Which would be best?

a) DIMENSIØN A(25)

b) DIMENSIØN A(N)

c) DIMENSIØN A(14)

d) DIMENSIØN A(15)

e) DIMENSIØN A(2)

f) DIMENSIØN A(3)

g) DIMENSIØN A

h) DIMENSIØN A(7,10,15)

2. If A is stored in location 11003 (octal), where in memory are A(0), A(4), A(19) and A(593) stored?

3. If X(39) is stored in location 10417 (octal), where in memory are X, X(0), X(8) and X(23) stored?

4. Which of the following declarations and sets of declarations are correct? Explain any errors. What is the value of each component of each vector if these statements would compile?

a) VECTØR VALUES A(K, 2) = 3.

b) VECTØR VALUES A(4) = 3, 6, 9

c) VECTØR VALUES PI = 3.14159, X = 2.69

d) VECTØR VALUES J = 0, 4.6, 9.2, 1.8

e)
```
VECTØR VALUES  B = 3.9, 6.4
NØRMAL MØDE IS INTEGER
DIMENSIØN B(5)
```

f)
```
VECTØR VALUES  C = 3, 4, 6
VECTØR VALUES  C(3) ... C(8) = 0
```

g) VECTØR VALUES D = 4.9, 1.8, 7.6, -4.3

h) VECTØR VALUES P = 1, 2, 3, 4, 5 *

i) VECTØR VALUES L(K) = 3.6, 4.1

j) VECTØR VALUES E = 2.7182818

5. Write MAD declarations which will declare the variables as required.

a) A in integer mode and having 21 elements A(0), ..., A(20).

b) B in Boolean mode and having 7 elements B(0), ..., B(6).

c) CAR in integer mode and set to an initial value of 54.

d) DØG a vector of 7 elements having initial values -1, 4, 3, -7, 13, 11, -5 and integer mode.

e) EASY an integer mode vector of 43 elements having EASY(0) set to 43.

f) FØX a floating point mode vector having initial values FØX(2) = 7.3, FØX(3) = 05.4, FØX(7) = 9.2 and FØX(11) = 1.9.

## 3.7 Basic Input-Output

The first category of MAD executable statements that we will describe in detail are those statements that are used to bring data into the computer and to print out results and comments. In this section, we will limit ourselves to only a few of the input-output statements, leaving for an appendix the discussion of formatting and more versatile input-output operations.

The two basic input statements that we will discuss are:

```
READ DATA
READ AND PRINT DATA
```

Before the execution of these statements can be discussed, it is necessary to specify the format in which data must be presented to the computer when these statements are used.

Definition of data variable, data item, and data set:

```
<data-var>    is   <var-name>  or  <var-name> ( L{ <deci-con> } )
<data-item>   is   <data-var>  =  L{ <constant>  or  <char-string> }
<data-set>    is   L{ <data-item> } *
```

Thus a data set consists of a list of data items separated by commas and terminated by an asterisk (*). In each of these definitions, the list often contains only one member (but is never empty). A further limitation on <data-item> is that the constants must be of such a mode that can be converted to the mode of the variable. Conversion between floating point and integer modes is always possible, but not between integer or floating point modes and Boolean mode. Examples of data items are:

```
X = 3
A(1) = -7.6E3
PQ = $ITEM$
I(0) = 1,2,-3,4,5
L = -3.9,2.654,.0000000000000000001603
K = $A$,$B$,$45$,$,$
PQ(6) = $THIS IS MØRE THAN ØNE CØNSTANT$
Y = -1,3.4,-2.9,5,0
M(2,3) = 4.6
```

A typical data set might be

```
X = 3,I = 1,2,-3,4,5, A(1) = 7.6*
```

Data sets may be punched anywhere in the first 72 columns of the data cards. The list may extend over more than one card and is terminated only by the asterisk,

not by the end of the card. The break between cards must occur just after the input variable name, just before or just after the equal sign, or between the elements in the list of constants and character strings. Commas in the lists which would occur as the last non-blank character on the card may be omitted. If a constant is split between two cards, it will be treated as two constants, as an illegal data item if the E of a floating point constant is the first character on a card, or as an otherwise illegal constant. Long character strings must be broken into multiples of six characters and made into two or more strings. If a variable name is split between cards it will usually be treated as an illegal data item. For example, if an attempt is made to read the following two cards

```
                    XR
          AY = 7.11*
```

the program will be terminated and an error message something like the following will be printed:

```
***** BAD CARD,              XR
***  XR     IS NØT A VARIABLE IN THE CALLING PRØGRAM
      AT LØCATIØN 10021
THE FIRST CARD NØT READ DURING EXECUTIØN WAS - -
     AY = 7.11*
          2 ADDITIØNAL CARDS WERE NØT READ.
```

The above basic input statements produce essentially the following actions. Since the exact behavior in all possibilities is quite complex, this description is somewhat abbreviated and assumes a valid data set.

1) An attempt is made to read the next data card image from the input tape. If there is no data card, the comment

```
****ALL INPUT DATA HAVE BEEN PROCESSED
```

is printed on the output listing and control is returned to the system which terminates the program.

2) If a data card is found, the first 72 columns on it are scanned from left to right. A direct copy of the card is also printed when the statement being obeyed is READ AND PRINT DATA.

3) When a variable name is found, its mode and address are obtained from the calling routine and the subscript(s), if any, used to compute the address of the input variable. The equal sign signals that constants or character strings are to follow.

4) The first constant or six characters of a character string is read, converted to the mode of the input variable (if necessary) and stored in the location of the input variable.

5) If the input variable is an array and additional constants or characters follow, they are read, converted (if necessary) and stored in successive locations of the array.

6) If the asterisk (*) terminating the data set is encountered, the reading operation is terminated with the input tape having been advanced to the beginning of the next data set, if any. Punches on a card after the terminal asterisk (*) are ignored and so each data set must begin on a new card.

7) If column 72 is reached without the data set terminating with an asterisk (*), the reading routine returns to step (1). Should the routine have been in the middle of the reading of a data item, this fact is preserved and the succeeding steps taken accordingly.

Although the READ AND PRINT DATA statement prints out the input so that the user has both his input and output listed on the output listing, it does not indicate how the computer may have interpreted his input cards. If there were errors in the form of input data which cause it to be read incorrectly this is not easily discovered. A better check is therefore obtained by "echo printing" the input using the PRINT RESULTS statement to be described below. This involves simply printing out all the input variables, but after they have been converted to machine form and stored.

To be more specific, consider what occurs when a data item involving a single variable is read. First, the mode of the variable contained in that item and the storage location reserved for that variable are obtained. The constant is converted to the mode of the variable, if necessary, and it is then stored in the location reserved for that variable. Thus, for the first example above, suppose the mode of X is floating point and the storage location reserved for it is 10127. Since the constant 3 is in integer mode, it must be converted to floating point mode before it is stored in location 10127. These same steps are taken if the data item contains a list of constants; however, each constant after the first is converted to the correct mode as necessary and stored in the memory location immediately preceding the one for the previous constant. Thus, if I(0) is an integer mode vector variable, in the fourth data item above, which is stored in 10124, then 1 is stored in 10124, 2 in 10123, -3 in 10122, 4 in 10121, and 5 in 10120. Thus, the data item

```
I(0) = 1, 2, -3, 4, 5
```

and the items

```
I(0) = 1, I(1) = 2, I(2) = -3, I(3) = 4, I(4) = 5
```

cause the identical constants to be stored in the same locations.

Several points about these input statements should be added. First, each constant in a data item should be written in the mode of the corresponding variable. This saves the time required for the conversion of the data. Second, floating point mode constants can be written in either form, i.e. with or without the power of ten. Also, very small or very large constants can be written out without the power of

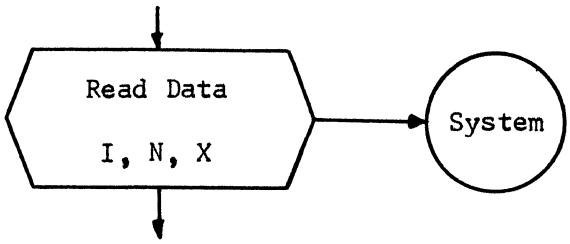
ten, such as .0000000000000000001603. These will be converted correctly on input. The use of more than eight digits of precision for an input constant is also legal, but will not result in any greater accuracy in the machine representation. Third, the MAD compiler ignores anything punched after READ DATA or READ AND PRINT DATA on that card (or continuation cards following it), so it is often helpful to someone reading the routine to have the names of the variables to be read appear in the statement, e.g.

```
READ AND PRINT DATA      I, N, X
```

Fourth, if a variable name appearing on a data card is not used in the routine (and thus no storage is reserved), that data set will be rejected, with the following diagnostic printed:

```
***** BAD CARD,  B = 1.2*
***  B      IS NOT A VARIABLE IN THE CALLING PROGRAM
       AT LOCATION 10020
```

The input statements are represented in a flowchart by the hexagonal subroutine box, since these statements actually call for the execution of subroutines.




The names of the variables to be read are usually included in the box. The normal path, executed when a set of data is found, is out the bottom. Otherwise, as is indicated, control is returned to the system.

Two classes of outputs are generally required. One of these, referred to in MAD as <u>results</u>, consists of the values of variables or expressions. For example, the value of x calculated as the solution of $ax + b = 0$ (when $a \neq 0$) is the result in the routine discussed in Chapter One. Equally important outputs are <u>comments</u>, which may contain the complete information on the solution of the problem (e.g. the comment in the linear equation routine of Chapter One that states "No value of x satisfies this equation") or may contain information about the results (e.g. a comment that the root is imaginary for this case and only the magnitude is being printed). In this section, we will consider three basic statements for the printing of results and one such statement for the printing of comments.

The three statements for the printing of results are:

PRINT RESULTS L{ <output-element> }
PRINT BCD RESULTS L{ <output-element> }
PRINT ØCTAL RESULTS L{ <output-element> }

where the nature of the elements which may be listed to be output must be defined. Definition of output element, output variable, output expression, and output block:

<output-element> is <output-var> or <output-exp> or <output-block>
<output-var> is <a-var> or <bool-var>
<output-exp> is <a-exp> or <bool-exp>
<output-block> is <output-var> ... <output-var> or
<output-var> ,..., <output-var>

We will first discuss the PRINT RESULTS statement and then the differences between this statement and the other statements for printing values. The output list specifies the variables, expressions, or blocks, the values of which are to be printed. If a single variable is specified, the name of the variable as well as the value of the subscript, if any, is printed followed by an equal sign and the value of that variable. If an expression (or, as a special case of an expression, a constant) is specified, three dots (...) are printed followed by the equal sign and the value of the expression. Outputs in these forms are printed four to a line, separated by commas. Each output item uses a maximum of 32 characters. The statement

```
PRINT RESULTS A, N+1, B(N+2)
```

might produce an output of the form

```
A =      2.000000,      ... =           14,    B(15) = 1.602000E-19
```

Note that this example has been abbreviated so that it will fit on the page. A more complete sample of this and other output formats is shown in Figure 3.1. A block in the output list causes the entire region of memory between and including the two indicated variables to be printed in the mode of the first. The values in an output block are printed 8 to a line preceded by a line giving the limits of the block and always begin on a separate line. As an example, the statement

```
PRINT RESULTS  P ... P(10)
```

produces the printout

```
          P(0) ... P(10)
     4         6        23         7       -26      4312         0         6
   213        -7      -164
```

Single outputs and blocks may be included in a single statement (see Figure 3.1).

The PRINT ØCTAL RESULTS statement produces printed results in the same format as PRINT RESULTS, except that the octal representation of the values of the variables or expressions are printed. Thus, using the same values of variables as above, the statement

```
PRINT ØCTAL RESULTS  A,P + 5
```

produces the printout

```
A = 202400000000,      ... = 000000000011
```

The PRINT BCD RESULTS statement allows the full character set to be printed. Each word is interpreted as BCD code and six characters printed accordingly. Thus, the two statements

```
PRINT ØCTAL RESULTS  R
PRINT BCD RESULTS  R
```

might produce the output

```
R = 222124600211
R =       BAD 29
```

The format for single outputs is the same for this statement as for PRINT RESULTS, i.e. four values to a line. However, no spaces are left between words when the block notation is used, thus allowing up to 131 characters per line. Consequently, the statement

```
PRINT BCD RESULTS  S...S(6)
```

might produce an output of the form

```
S(0)...S(6)
   THIS IS AN EXAMPLE ØF A BLØCK PRINTØUT.
```

The output statement for printing comments is of the form:

PRINT CØMMENT {$ <c-c-c> $\underline{S}_0^{131}$ { {<character> not $ } or $$ } $ or $$}

where we must give the

Definition of carriage control character:

<c-c-c> is + or <blank> or 0 or - or 1 or 2 or 4 or 6 or 8 or 9

This first character of a line to be printed out is used to specify the vertical spacing of the output. The carriage is that part of the printer which moves the paper through the machine past the printing positions. The carriage control character is used to determine the motion of the carriage to be made before the line is printed and thus allow single spacing, double spacing, skipping to the top of the next page and other such actions to precede the printing. Table 3.4 shows the meaning of the carriage control characters listed above under some operating systems.

Table 3.4 Carriage Control Characters

| Character | Carriage Action |
|---|---|
| + | No space--overprint the previous line |
| blank | Single space before printing |
| 0 | Double space before printing |
| - | Triple space before printing |
| 1 | Skip to top of next page before printing |
| 2 | Skip to next half-page before printing |
| 4 | Skip to next quarter-page before printing |
| 6 | Skip to next sixth-page before printing |
| 8 | Skip to next sixth-page before printing |
| 9 | Single space before printing suppressing overflow control, i.e. print across page boundaries without skipping |

In other systems, the carriage is automatically positioned to print on the next line after printing each line unless a + is used (in which case no advance takes place). The blank causes no additional advancement (so that the output is single spaced), the zero causes a single space before printing in addition to the one after printing and the other characters cause essentially the same action described in Table 3.4.

The printing system is designed so that sixty lines will normally be printed per page whereas there are 66 lines possible on a page. This arrangement automatically produces a small margin at the top and bottom of each page. When the carriage control character "1" is used the machine actually skips to the next page and skips the first three lines at the top of that page. When sixty lines have been printed or skipped on a page the machine automatically skips to the top of the next page unless carriage control character "9" is being used. If the carriage is on the bottom half of a page and a carriage control character "2" is used, the machine actually skips to the top of the next page. The carriage control character itself is not printed. If a character not defined as a carriage control character is used as the first character of a line to be printed, the program usually single spaces before printing and does not print the offending character. In some systems a special

character (e.g. ≠) is printed in place of the offending character to signal the user that he has used an illegal carriage control character.

Two final points should be made about the PRINT CØMMENT statement. First, the statement causes the exact contents between the $'s to be printed with the exception of the carriage control character and the $$ representation for $. Thus the statement

```
PRINT CØMMENT $ THE LØØP WAS EXECUTED N TIMES.$
```

will result in exactly the statement

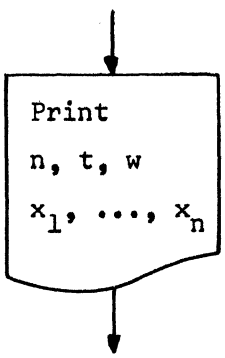
```
THE LØØP WAS EXECUTED N TIMES.
```

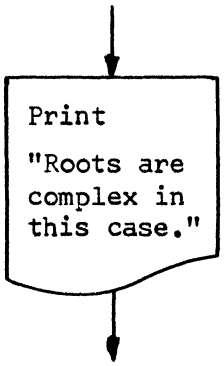
being printed; the value of N is not substituted in the comment. Two $'s are used here to represent a single $ in the output just as in character constants. Second, if no comment is to be printed (i.e. the statement is to be used only to advance the carriage), the carriage control character may be omitted if a blank is intended. Thus,

```
PRINT CØMMENT $$
```

would result in the carriage being advanced one line and nothing being printed.

The basic output statements are represented in flowcharts by a box shaped like a torn-off sheet of paper. The variables whose values are to be printed or the comment which is to be printed is usually included in the box.




In conclusion, we give a fairly complete demonstration of the basic input and output statements which have been introduced in this section. Figure 3.1a contains a MAD program with input data and Figure 3.1b illustrates the output listing produced by this program.

```
$COMPILE MAD, EXECUTE
          R     DEMONSTRATION OF BASIC INPUT AND OUTPUT IN MAD.

           INTEGER DECI, OCTA, CHAR, INT(15), TEXT(12)
           BOOLEAN BOOL
           DIMENSION FPT(11), ARRAY(2*4)
           PRINT COMMENT $1$

          R    FIRST, INPUT AND OUTPUT OF THE BASIC CONSTANTS.
           READ AND PRINT DATA
           PRINT RESULTS DECI, OCTA, CHAR, BOOL
           PRINT OCTAL RESULTS DECI, OCTA, CHAR, BOOL
           PRINT BCD RESULTS DECI, OCTA, CHAR, BOOL
           PRINT RESULTS FPT1, FPT2, LARGE, SMALL
           PRINT OCTAL RESULTS FPT1, FPT2, LARGE, SMALL

          R    SECOND, THE PRINTING OF VECTORS AND ARRAYS.
           READ DATA
           PRINT RESULTS INT...INT(15), FPT(0),...,FPT(11)
           PRINT RESULTS ARRAY(1,1)...ARRAY(2,4)

          R    THIRD, THE PRINTING OF CHARACTER STRINGS.
           READ DATA TEXT
           PRINT BCD RESULTS TEXT(0)...TEXT(12)

          R    FOURTH, THE PRINTING OF VALUES OF SUBSCRIPTED VARIABLES.
           PRINT RESULTS INT(6), FPT(INT(2)), INT(FPT(10)),
          1     INT(DECI-18), ARRAY(2,3), FPT(FPT1), INT(1.37)

          R    FIFTH, THE PRINTING OF VALUES OF EXPRESSIONS.
           PRINT RESULTS 3.14159265, FPT1+2.63, FPT2.P.2, SQRT.(FPT1)

          R    SIXTH, SOME MISCELLANEOUS PRINTING.
           PRINT RESULTS FPT2, 1E-27, SQRT.(2.0), FPT(9)...FPT(5),
          1              2.0*DECI-OCTA, CHAR+FPT1, ARRAY(1.3,2.7)

          R    FINALLY, TO PUT A TITLE ON THE TABLE.
           PRINT COMMENT $0$
           PRINT COMMENT $0                                              FIGU
          1RE 3.1B  DEMONSTRATION OF MAD OUTPUT LISTING$
           END OF PROGRAM
$DATA
DECI = 25, OCTA = 25K, CHAR = $     C$, FPT1 = 9.0, FPT2 = 1.2075E3,
LARGE = 2.175E21, SMALL = .000000001604, BOOL = 1B *
INT = 2,-13,4,7,269,4096,-231,17,32,-41062,+69,-1,0,-0
FPT(2) = 2.7,3.6,-4.9,6.8,1.23E5,2.794E-12,0.0,-0.0,5.43,-1.7
ARRAY = 0.,1.,-1.,2.,-2.,3.,-3.,4.,-4. *
TEXT = $THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.$, $0123456789$,
       $+-*/=()$, TEXT(12) = $.,$$$ *
```

Figure 3.1a Input-Output Demonstration Program

```
DECI = 25, OCTA = 25K, CHAR = $     C$, FPT1 = 9.0, FPT2 = 1.2075E3,

LARGE = 2.175E21, SMALL = .000000001604, BOOL = 1B *

        DECI =            25,           OCTA =            21,           CHAR = -17997958163,           BOOL =            1B
        DECI =  000000000031,           OCTA =  000000000025,           CHAR =  606060606023,           BOOL =  000000000001
        DECI =        00001,            OCTA =        00000E,           CHAR =             C,           BOOL =        000001
        FPT1 =      9.000000,           FPT2 =   1207.500000,           LARGE =  2.175000E 21,         SMALL =  1.604000E-09
        FPT1 =  204440000000,           FPT2 =  213455700000,           LARGE =  307727501404,          SMALL =  143670716734

              INT(0)...INT(15)

         2            -13             4              7            269           4096           -231             17
        32         -41062            69             -1              0             -0     1006766073     1006766073

              FPT(0)...FPT(11)

1.884522E-37   1.884522E-37   2.700000E 00   3.600000E 00  -4.900000E 00   6.800000E 00   1.230000E 05   2.794000E-12
 .000000E 00   -.000000E 00   5.430000E 00  -1.700000E 00

              ARRAY(1,1)...ARRAY(2,4)

1.000000E 00  -1.000000E 00   2.000000E 00  -2.000000E 00

3.000000E 00  -3.000000E 00   4.000000E 00  -4.000000E 00

              TEXT(0)...TEXT(12)

  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.    0123456789  +-*/=()     .,$

      INT(6) =          -231,         FPT(4) =     -4.900000,         INT(5) =          4096,         INT(7) =            17

   ARRAY(2,3) =      4.000000,         FPT(9) =      -.000000,         INT(1) =           -13

         ... =      3.141593,            ... =     11.630000,            ... = 1.458056E 06,            ... =      3.000000

        FPT2 =   1207.500000,            ... = 10.000000E-28,            ... =      1.414214

              FPT(9)...FPT(5)

-.000000E 00    .000000E 00   2.794000E-12   1.230000E 05   6.800000E 00

         ... =     29.000000,            ... = -2.045218E 08,      ARRAY(1,2) =     -1.000000
```

FIGURE 3.1B DEMONSTRATION OF MAD OUTPUT LISTING

Exercises

1. Which of the following instructions are correct? What is wrong with the others?

a) PRINT RESULT X

b) READ DATA

c) READ AND PRINT X,Y

d) PRINT CØMMENT $NØ ANSWER $

e) PRINT RESULTS A, B(0) ... B(7),

f) PRINT RESULTS -A + 3*J, B(I)

g) PRINT RESULTS A(3), ... A(7), XYZ(9)

h) PRINT RESULTS X1, ..., X7

i) PRINT CØMMENT $0 THE PRICE IS $9.98.$

2. What, if anything, is wrong with the following data cards or set of cards?

a) X(0) ... X(3) = 1, 2, 3, 4*

b) X(0) = 1
X(1) = 2
X(2) = 3
X(3) = 4*

c) X = 3., I = 1, Y(I) = 1, 7, -4, 3*

d) PI = 3.14159

e) A = B - 3.0, C = D = 0

3. In each of the following, a READ DATA instruction is used to read the data card indicated. The mode of X, Y and Z is assumed to be floating point and A, M and N are assumed to be in integer mode. The output instructions listed are then performed. What is the output?

a) X = 1, X(1) = 2, X(2) = 7, 8, 9, 10, Y = 2.*
PRINT RESULTS X(0) ... X(5), X.P.2 - Y

b) A(0) = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, Z = 13.2*
PRINT RESULTS A, A(1), A(2), A(3), A(4), A(5)
PRINT RESULTS A(0) ... A(5), Z, 3
PRINT RESULTS A(0) ... A(10)

c) M(2) = 5, N = 1, 2, 3, 4, M = 7, 6, 8*
PRINT RESULTS M(N)
PRINT RESULTS N(1) ... N(3), M(0), M, M(1), M(2)

# Chapter Four

## BASIC ARITHMETIC PROCESSES

Up to this point we have developed only sufficient language to bring data into the computer and to print out results and comments. A large portion of the manipulations performed on that data, both numeric and non-numeric, are referred to as arithmetic in MAD. In this chapter, we will present those parts of the MAD language that enable us to perform arithmetic on the data.

We will first describe the arithmetic operations and the MAD symbols used to signify these operations. Arithmetic expressions vary in complexity from those that are merely single variables or constants to those involving many operations. The rules governing the construction of such expressions are presented next. The assignment or substitution statement is used to set the value of a variable equal to that of an arithmetic expression.

With this capability developed, rather complex processing of data becomes possible. The true power of the digital computer, however, will not be realized until branching and repetitive processes are introduced in the next two chapters.

Although this chapter is devoted primarily to the processing of numeric data, we will introduce some of the tools of non-numeric processing. Since all data are represented numerically within the computer, it is not surprising that many of the operations on this data have been included in the MAD language as part of arithmetic expressions.

### 4.1 Arithmetic Operations

In this section we will first define two classes of arithmetic operations and then develop that portion of the MAD language which is used to designate these operations. A unary operation on a set of elements is a rule which assigns to each element from that set a uniquely defined element from that same set. Mathematically, an operation requires only that such a rule be defined, whereas in the computer, an operation requires that the element defined by the rule, the result, be produced. The original element from the set, on which the operation is defined, is called the operand. An example of a unary operation is integer negation, where the operand is X, an integer, and the result is then -X, also an integer. Many of the more common operations require two operands and are thus called binary.

This use of the word binary should not be confused with the use meaning base 2; the sense of binary will generally be clear from the context. A binary operation on a set of elements is a rule which assigns to each ordered pair of elements from that set a uniquely defined third element from that set. In this section, we will define operations on two different sets of elements, namely the integer mode numbers and the floating point mode numbers. Because of the limitations of computers, both of these sets are finite. In a later section, we will define operations which are applied independently to all of the bits of a machine word in parallel. In the next chapter, operations on the set of Boolean values are introduced.

There are three unary arithmetic operators and five binary arithmetic operators in MAD. Each of the operations designated by these operators are defined for both integer mode and floating point mode operands. Table 4.1 shows the three unary operators and a pair of equivalent MAD and algebraic expressions. The unary operator + may be used, but causes no operation to be performed. These operations, in either

Table 4.1 Unary arithmetic operators

| Operator | Operation | MAD Expression | Algebraic Expression |
|---|---|---|---|
| + | none | +X | x or +x |
| - | negation | -X | -x |
| .ABS. | absolute value | .ABS. X | \|x\| |

integer mode or floating point mode, operate on the sign bit of the operand. The operation of negation changes the sign bit, and the absolute value operation sets the sign bit to zero (plus). A metavariable for these unary arithmetic operations will be needed in other definitions and is given now.

Definition of unary arithmetic operator:

<un-a-op> is + or - or .ABS.

The five binary arithmetic operators are now also defined formally and assigned a metavariable for future use.

Definition of binary arithmetic operator:

<bin-a-op> is + or - or * or / or .P.

These operators, along with equivalent expressions involving them as written in MAD and the usual algebraic notation, are shown in Table 4.2.

Table 4.2 Binary arithmetic operators

| Operator | Operation | MAD Expression | Algebraic Expression |
|---|---|---|---|
| + | addition | A + B | a + b |
| - | subtraction | A - B | a - b |
| * | multiplication | A * B | ab or a·b |
| / | division | A / B | $\frac{a}{b}$ |
| .P. | exponentiation | A.P.B | $a^b$ |

Note that the asterisk (*) must always be used to denote multiplication in MAD. Although xy and u(v+w) are common algebraic notation, XY would be interpreted as a single variable name and U(V+W) would be interpreted as $u_{v+w}$. Each of the binary arithmetic operations is defined in integer mode (i.e. when both operands and the result are in integer mode) and in floating point mode.

The operations of addition, subtraction, multiplication and division in a computer were discussed in Chapter Two and thus require no additional comment here. The integer mode and floating point mode exponentiation are both compound operations in the computer, each requiring multiple steps. Integer mode exponentiation is accomplished by successive multiplications and, thus, 5.P.4 is interpreted as 5 * 5 * 5 * 5. Since this is a series of multiplications, the value of the first operand may be negative, producing a positive result if the exponent is even and a negative result if the exponent is odd. Although the second operand may be negative, an integer (other than +1 or -1) raised to a negative integer power is zero. Floating point mode exponentiation, i.e., the operation when both operands and the result are in floating point mode, is evaluated using the identity

$$a^x = e^{x \ln a}$$

Since this operation requires the calculation of the logarithm of the first operand, a, that operand must be greater than zero. This restriction is consistent with mathematical considerations and avoids any problem of producing complex results. Thus, to raise a negative floating point number to an integral power, it is necessary to make clear that the power is an integer. In MAD, it is therefore required that such a power be expressed in integer mode and this case of exponentiation is treated separately. This third form of the exponentiation operation, the raising of a floating point mode number to an integer power, is not strictly an operation in

the mathematical sense, since the two operands are members of different sets. Following normal mathematical notation, however, the MAD operator .P., with the first operand a floating point mode number and the second an integer, results in the execution of floating point mode multiplications (followed by an inversion if the power is negative). Thus,

```
(-3.6) .P. 4   =   (-3.6) * (-3.6) * (-3.6) * (-3.6)
(1.5) .P. (-2)   =   1. / (1.5 * 1.5)
```

Each of the above operations except exponentiation is defined only when the modes of the operands agree. If two operands of different modes are to be added, subtracted, multiplied or divided or if an integer is to be raised to a floating point power, the integer is always converted to floating point mode and the floating point mode operation is executed. Examples of this are:

```
3 / 2.  =  3. / 2.  =  1.500000
3. / 2  =  3. / 2.  =  1.500000
4 + 0.  =  4. + 0.  =  4.000000
4.P..5  =  4..P..5  =  2.000000
```

Execution of a program may be terminated and diagnostics produced due to an attempt to perform an arithmetic operation which would produce an invalid result. In each case, a message describing the error is printed, followed by the statement[1]

```
**** ERROR RETURN TO SYSTEM.
```

Some of the errors are described below. First, if any operation, other than floating point mode exponentiation, produces results too large in magnitude for the machine to represent, overflow occurs. The resulting diagnostic is of the form

```
**** OVERFLOW AT LOC 10317
```

where the location in memory of the instruction being executed when the overflow occurred is specified. Floating point operations may produce non-zero results that are too small in magnitude for the machine (e.g. $10^{-20} * 10^{-25} = 10^{-45}$). This condition is referred to as underflow,[2] and produces a diagnostic similar to that for overflow, namely

```
*** UNDERFLOW AT LOC 12345
```

---

[1] The exact form of the diagnostic may vary from one operating system to another. We will show some typical diagnostics.

[2] It is often desired to set the result to zero when underflow occurs. Subroutines to do this, as well as to handle overflow, are available in MAD. See the descriptions of FTRAP. and FSPILL. in the Appendix D.

Two types of diagnostics that are produced during the execution of floating point mode exponentiation are

```
**** EXP2 NEGATIVE EXPONENT
        AT LOCATION 10316
```

when the value of the first operand is negative and

```
**** EXP1 ANS TOO LARGE
        AT LOCATION 10076
```

when the result that would be produced is too large. This latter case is treated as overflow when the second operand is integer mode.

Finally, an error that goes undetected is division by zero. The result produced is equal to the dividend for integer mode division and is zero (independent of the value of the dividend) for floating point mode division. Unless special attention is given to the problem of division by zero, completely erroneous results may be produced, with no indication of the error given.

## 4.2 Arithmetic Expressions

Before we are able to describe the formation of arithmetic expressions, we need to discuss the use of parentheses in MAD. Parentheses may be used in the same manner as in algebraic notation. Redundant parentheses are allowed, e.g. ((A) + ((B))). Whenever parentheses are used, they must be used in pairs, i.e. there must always be a right parenthesis, ), corresponding to and following each left parenthesis, (, as well as a left parenthesis corresponding to and preceding each right parenthesis. Each pair of parentheses must enclose an arithmetic expression.

The definition of an arithmetic expression can now be given recursively in two forms. In words, an arithmetic expression is one of the following:

1) an arithmetic constant (integer or floating point mode);
2) an arithmetic variable (integer or floating point mode);
3) a function call which returns an integer or floating point mode result, <a-fn-call> (See Chapter 7);
4) an integer mode word produced by bitwise operations, <bit-word> (See Section 4.4);
5) any arithmetic expression enclosed in parentheses;
6) any arithmetic expression preceded by a unary arithmetic operator;
7) any two arithmetic expressions connected by a binary arithmetic operator.

Any arithmetic expression can be constructed by repeated application of these rules and anything that can be obtained by repeated application of these rules is an arithmetic expression. The metalanguage definition is now given.

Definition of an arithmetic expression:

<a-exp> is <a-con> or <a-var> or <a-fn-call># or <bit-word>#
or (<a-exp>) or <un-a-op> <a-exp>
or <a-exp> <bin-a-op> <a-exp>

Some examples of arithmetic expressions are given in Table 4.3.

Table 4.3 Examples of arithmetic expressions

| Reference Number | Expression | Type of Expression |
|---|---|---|
| 1 | 6.9 | a floating point mode constant |
| 2 | 7 | an integer mode constant |
| 3 | AX2 | a variable |
| 4 | A(7) | a subscripted variable |
| 5 | P | a variable |
| 6 | -AX2 | unary operator acting on expression 3 |
| 7 | .ABS. A(7) | unary operator acting on expression 4 |
| 8 | P + 7 | binary operator connecting expressions 5 and 2 |
| 9 | P - AX2 | binary operator connecting expressions 5 and 3 |
| 10 | (P + 7) | expression 8 enclosed in parentheses |
| 11 | AX2 * (P + 7) | binary operator connecting expressions 3 and 10 |
| 12 | AX2 * (P+7) / .ABS. A(7) | binary operator connecting expressions 11 and 7 |

In each of the examples except the last, sufficient parentheses were used so that the order in which operations were to be performed was obvious. Knowing the order in which the computer performs the operations is critical, since for example, the expression

3. / 2. * 4.

could be interpreted as

(3. / 2.) * 4.

in which case the answer is 6. , or as

3. / (2. * 4.)

in which case the answer is .375. With most computer languages, including MAD, the first of these interpretations is chosen.

Let us now examine the method used in MAD for the evaluation of arithmetic expressions. Each expression is broken into a set of elementary expressions, i.e. those expressions consisting of a single operator (and one or two operands). These elementary expressions are then evaluated sequentially. In the interest of efficiency, the instructions produced by the MAD compiler will sometimes execute the computations in an order different from that indicated here. However, the results produced are identical to those indicated below (including any effect of roundoff). We will not indicate, when discussing the evaluation of arithmetic expressions, the steps taken by the computer to convert the mode of an operand when the modes of the operands of a binary operator do not agree.

When breaking down an expression, the compiler scans the expression from left to right. It first takes all expressions enclosed in parentheses, taking the innermost parentheses first. Thus, the expression

A * (B + (C / D))

would be broken into elementary expressions

$T_1 = C/D$

$T_2 = B + T_1$

$T_3 = A * T_2$

where the $T_i$ are names used to label an intermediate result. These temporary results may sometimes be handled completely in the registers, but often some of them must be stored temporarily in memory. The order in which operands are taken when parentheses are not involved is determined by the precedence list shown in Table 4.4.

Table 4.4 Arithmetic operator precedence list

| | |
|---|---|
| .ABS. | + (unary) |
| .P. | |
| - (unary) | |
| * | / |
| + (binary) | - (binary) |

Operators of higher precedence are always performed before those of lower precedence. Thus, the parentheses around C/D in the previous example are unnecessary. Finally, when more than one operator on the same line on the precedence table (e.g. two divisions or a multiplication and a division) are involved, they are taken in left to right order. As an example consider the following expression.

$$-A * B + C - D + - E * F$$

The elementary operations would be

$$T_1 = -A$$

$$T_2 = -E$$

$$T_3 = T_1 * B$$

$$T_4 = T_2 * F$$

$$T_5 = T_3 + C$$

$$T_6 = T_5 - D$$

$$T_7 = T_6 + T_4$$

The MAD compiler reorders these operations so as to reduce the number of times the temporary results must be stored. Note that whenever a temporary result is not used in the next operation ($T_1$, $T_2$, $T_3$, and $T_4$ in this example), it must be stored. The storage and the subsequent retrieval of the result often waste both time and storage space. Thus, the compiler orders these elementary expressions as follows:

$$T_2 = -E$$

$$T_4 = T_2 * F$$

$$T_1 = -A$$

$$T_3 = T_1 * B$$

$$T_5 = T_3 + C$$

$$T_6 = T_5 - D$$

$$T_7 = T_6 + T_4$$

Now, only $T_4$ is not used in the next operation and must be stored. Notice that exactly the same operations on the same operands are performed in both evaluations; thus, exactly the same results are produced. Some applications of the rules of precedence are shown in the examples of Table 4.5.

Table 4.5 Examples of the Application of the Rules of Precedence

| MAD | MAD (Parentheses Added) | Algebraic Notation |
|---|---|---|
| A + B * C | A + (B * C) | $a + bc$ |
| -A + B * C | (-A) + (B * C) | $-a + bc$ |
| -A * B + C | ((-A) * B) + C | $-ab + c$ |
| A * B - C | (A * B) - C | $ab - c$ |
| A * + B - C | (A * ( +B)) - C | $ab - c$ |
| A * B + - C | (A * B) + (-C) | $ab + (-c)$ |
| .ABS.X + Y | (.ABS.X) + Y | $\|x\| + y$ |
| .ABS. X .P. Y | (.ABS. X) .P. Y | $\|X\|^{y}$ |
| X.P. .ABS. Y | X .P. (.ABS. Y) | $x^{\|y\|}$ |
| ——— | .ABS. (X.P.Y) | $\|x^{y}\|$ |
| X .P. -Y | X .P. (-Y) | $x^{-y}$ |
| -X .P. Y | -(X .P. Y) | $-x^{y}$ |
| ——— | (-X) .P. Y | $(-x)^{y}$ |
| X .P. Y + Z | (X .P. Y) + Z | $x^{y} + z$ |
| A / B / C | (A / B) / C | $\frac{a}{bc}$ |
| ——— | A / (B * C) | $\frac{a}{bc}$ |
| A / B * C | (A / B) * C | $\frac{a}{b} \cdot c$ |
| A * B / C | (A * B) / C | $\frac{ab}{c}$ |
| A * B / C * D | ((A * B) / C) * D | $\frac{ab}{c} \cdot d$ |
| ——— | (A * B) / (C * D) | $\frac{ab}{cd}$ |
| A / B + C * D - E | ((A / B) + (C * D)) - E | $\frac{a}{b} + cd - e$ |
| A + B / C + D | (A + (B / C)) + D | $a + \frac{b}{c} + d$ |
| ——— | (A + B) / (C + D) | $\frac{a + b}{c + d}$ |

The next consideration is the mode of each operation and finally the mode of the expression. The mode in which each operation is performed is determined solely by the mode of the operand or operands of that operation. Thus, in the last example, if A, B, C and E are in integer mode and D and F in floating point mode, the following takes place:

1) $T_2$ is calculated in integer mode.
2) $T_2$ is converted to floating point mode.
3) $T_4$ is calculated in floating point mode.
4) $T_1$ is calculated in integer mode.
5) $T_3$ is calculated in integer mode.
6) $T_5$ is calculated in integer mode.
7) $T_5$ is converted to floating point mode.
8) $T_6$ is calculated in floating point mode.
9) $T_7$ is calculated in floating point mode.

The mode of the expression is always the mode of the last operation. This mode is integer if and only if all constants, variables and functions in that expression (other than subscripts and function arguments) are in integer mode and is floating point if any constant, variable or function (other than subscripts and function arguments) in that expression is in floating point mode. As a final example the expression

```
4. + 3 / 2 * 2
```

is evaluated as

```
T1 = 3 / 2
T2 = T1 * 2
T3 = 4. + T2
```

Since 3 and 2 are both integer mode constants, $T_1$ is calculated, using integer mode division, as 1 (integer mode). $T_2$ is also calculated in integer mode (as 2). Finally, $T_2$ is converted to floating point mode before being added to 4., producing the final result, 6., in floating point mode.

In order to make it possible to refer to expressions whose value has a particular mode, we define some additional metavariables.

Definitions of integer and floating point expression:

<int-exp> is an arithmetic expression of integer mode

<f-p-exp> is an arithmetic expression of floating point mode

Exercises

1. Write an algebraic expression corresponding to each of the following MAD expressions.

a) MN

b) A*B.P.C*D/E*F + G

c) .ABS.A.P.2

d) .ABS.(A.P.2)

e) (.ABS.A).P.2

f) -A.P.3*I

g) -(A.P.3*I)

h) -A.P.(3*I)

i) -(A.P.(3*I))

j) -(A.P.3)*I

k) A*B + C*D

l) A-B*X + C*X*X

m) A-B*X + C*X.P.2

2. Write MAD expressions for each of the following algebraic expressions. (Note: All variables and constants other than exponents and subscripts are in floating point mode. Exponents may be integers.)

a) $ax^2 + bx + c$

b) $|b^2 - 4ac|$

c) $\frac{ab + cx}{3d - 4x}$

d) $\sum_{i=1}^{4} x_i$

e) $\frac{4x^2 + 3}{3x^3 + 2x^2 + x - 8}$

f) $(a^2 + b^2)^{1/2}$

3. Which of the following are valid arithmetic expressions? Indicate what is wrong with those that are not. Evaluate those that are valid using A = 1., B = 2., C = 3., and D = -2. .

a) A / B

b) A(B + D)

c) A * (B + D)

d) .ABS. (A - (B * C)

e) D .P. 2

f) $\frac{A}{2C}$

g) - B * - D

h) B + .ABS. D

i) ((A + B) .P. (A / (- D)))

j) (A + (B)) / (C + D)) - ((A + (3 * B)) - (2 * C + D)

4. Evaluate each of the following expressions if A = 1.0, B = 3.0, C = 0.5, and D = 10.0.

a) 2.*A*B/C

b) (-A+D).P.C + B

c) .ABS.A+B

d) A*B/C*D

e) -A*B + C

5. If A is an integer, what is the value of (A/2)*2 for different values of A?

6. If A is an integer, what is the value of (A*2)/2 for different values of A?

7. Evaluate, writing your answers in the correct mode and showing all intermediate results.

a) 4/3*2 + 1.

b) 7. + 1/3

c) 3.P.2 - 8/(2+2.)

d) 9 .P. .5

e) 1 - 9/10

f) (3 + 0.)/4

## 4.3 The Arithmetic Substitution Statement

When the value of an arithmetic expression is to be calculated and stored for future use, the substitution or assignment statement is utilized. The arithmetic substitution statement[3] is of the form:

<a-var> = <a-exp>

The execution of this statement is as follows:

1) The arithmetic expression is evaluated, yielding a numeric result.
2) If the mode of the expression differs from the mode of the variable on the left, the result is converted to the mode of that variable.
3) The variable is then set equal to the value of the expression.

Some examples of substitution statements are:

```
X(3) = 0
I = 1
P = $A WØRD$
```

where, in each case, the value of the variable is set equal to the constant on the right. If X is a floating point mode variable and I and P are integer mode variables, no conversion need take place. Another example of an arithmetic substitution statement is

```
I = 3.5 + 6/4
```

where I is an integer mode variable. The expression is evaluated as 3.5 + 1 = 3.5 + 1. = 4.5 (floating point mode). The result is converted to integer mode, 4, and I is thus set equal to 4 (integer mode). Note that in the evaluation of the expression 6/4 is computed in integer mode, even though the mode of the expression is floating point. If either 6 or 4 (or both) had been floating point mode, then the division would have been performed in floating point mode, and I would have been given 5 as its value. A final example of an arithmetic substitution statement is:

```
I = I + 1
```

If I is an integer mode variable which has value 7 before the execution of this statement, the expression would be evaluated as 7 + 1 = 8 and I would then be set equal to 8 in integer mode. In general, this statement increments the value of I by 1 (i.e. adds 1 to I) each time it is executed.

---

[3] The Boolean substitution statement is discussed in Chapter 5.

When a constant (such as π) is to be used many times in a program, it is sometimes convenient to give it a variable name by means of an assignment statement. Thus, execution of the statement

```
PI = 3.14159265
```

would allow PI to be written instead of the longer constant in all statements which are executed after that one. Even better in some ways is the use of a VECTØR VALUES statement for this purpose. The statement

```
VECTØR VALUES PI = 3.14159265
```

establishes the value of PI at compilation time and thus this statement can occur anywhere in the routine.

Now that we have seen some correct substitution statements, let us consider some illegal ones. The following represent illegal attempts at writing substitution statements.

```
X - Y = 0
3 = B
S = Y = 1
A(0)...A(5) = 0
M = 4, N = 3
P = $TØØ LØNG$
L = 4.6, 3.9, 2.1
```

In the first two cases, the error is that the term to the left of the equal sign is not a variable. In the remaining examples, attempts were made to include more than one substitution statement in a single statement. Although the last three lines are legal in data sets, none of these examples are legal substitution statements.

## Exercises

1. Indicate what, if anything is wrong with each of the following substitution statements:

   a) `X(I + 1) = X(I) - 2 * A`

   b) `A = B  C = 1`

   c) `3X = 21`

   d) `X(I) = X(X)`

   e) `5. * A = 5. - A`

   f) `I = 1/2 * 3 - 4 + 0`

   g) `F(X) = 0`

2. I, PQ, and INDEX are in integer mode and all other variables are in floatpoint mode; evaluate the variable at the left of the equal sign at each step,

assuming these instructions are performed in order.

```
I = 1
J = 2 * I/3 + 6
I = I + 1
MØNEY = 6.3 * J/2.0
PQ = MØNEY
I = I + 1
PQ = I.P.I
INDEX = 3
INDEX = INDEX/2 * 2
INDEX = INDEX/2 * 2
```

## 4.4 Bitwise Operations[4]

Six operators are available in the MAD language which call for manipulations on the bits of a word. The operands of these operators (and consequently the results) are always integer mode; an attempt to use a floating point mode operand will result in a compilation error. In all cases, the operands and the result are treated as 36 bit words with the sign bit having no special significance. One of the bitwise operators is the unary operator, .N., which represents the operation of complementing (sometimes referred to as not or negation). Each bit of the word on which it operates is changed, i.e., each 0 becomes a 1 and each 1 becomes a 0. The remaining five operators are binary and include the "and", "or" and "exclusive or" of logic. The truth tables shown in Table 4.6 indicate the effect of these operators, where $P_i$ is the i-th bit of the first operand and $Q_i$ is the i-th bit of the second operand. The values indicated by this table are obtained for all 36 bits.

Table 4.6 Truth Table for Binary Bitwise Operators

| $P_i$ | $Q_i$ | $(P \text{ .A. } Q)_i$ | $(P \text{ .V. } Q)_i$ | $(P \text{ .EV. } Q)_i$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

[4] This section may be omitted without loss of continuity.

The last two bitwise operations are shifts, allowing the shifting of the word to the left (.LS.) or to the right (.RS.). The first operand is the word to be shifted and the second operand is the number of bits to be shifted. Thus, A.RS.7 shifts A to the right by 7 places. The right seven bits of A are lost and 7 zeroes are added on the left. Equivalently, on a left shift, the left bits are lost and zeroes are added on the right. Shifts of more than 35 in either direction set the word to zero. Note that a left shift of i is equivalent to multiplying by $2^i$ if no ones are shifted out and if a 1 bit is not shifted into the sign bit position. Similarly, a right shift of a positive integer is equivalent to integer division by a power of 2. For negative integers, floating point mode numbers, and when ones are shifted off the left or into the sign bit, no equivalence between shifting and arithmetic operations exists.

The bitwise operators may now be defined formally for later use in the definition of a bit-word.

Definition of unary and binary bitwise operator:

<un-bit-op> is .N.

<bin-bit-op> is .A. or .V. or .EV. or .LS. or .RS.

Example

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 010 | 111 | 010 | 110 | 100 | 001 | 000 | 101 | 011 | 111 | 000 | 101 |
| B | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 | 101 | 110 | 111 | 010 |
| .N.A | 101 | 000 | 101 | 001 | 011 | 110 | 111 | 010 | 100 | 000 | 111 | 010 |
| A.A.B | 000 | 001 | 010 | 010 | 100 | 001 | 000 | 101 | 001 | 110 | 000 | 000 |
| A.V.B | 010 | 111 | 010 | 111 | 100 | 101 | 110 | 111 | 111 | 111 | 111 | 111 |
| A.EV.B | 010 | 110 | 000 | 101 | 000 | 100 | 110 | 010 | 110 | 001 | 111 | 111 |
| A.RS.8 | 000 | 000 | 000 | 101 | 110 | 101 | 101 | 000 | 010 | 001 | 010 | 111 |
| B.LS.21 | 111 | 101 | 110 | 111 | 010 | 000 | 000 | 000 | 000 | 000 | 000 | 000 |

Words which are produced by these bitwise operations were treated separately in section 4.2 when arithmetic expressions were defined. The formal definition of these words can now be given.

Definition of bitwise produced word:

<bit-word> is <int-exp> or <un-bit-op> <bit-word> or
<bit-word> <bin-bit-op> <bit-word>

Examples of members of <bit-word> are:

```
I .A. (J.LS.K)
.N.(P.V.Q)
(2 + I * J) .LS. (2 * I)
```

The bitwise operators are included in the precedence list as shown in Table 4.7.

Table 4.7 Precedence List of Arithmetic and Bitwise Operators

| | | | | |
|---|---|---|---|---|
| .ABS. | + (unary) | .N. | .LS. | .RS. |
| .A. | | | | |
| .V. | .EV. | | | |
| .P. | | | | |
| - (unary) | | | | |
| * | | | | |
| + (binary) | - (binary) | | | |

Exercises

1. Which of the following are legal MAD arithmetic expressions? Indicate the errors in each of the expressions, assume A, B, C, D and E are floating point mode variables and F, G, H, I and J are integer mode variables.

a) A + (H .LS. 3) * F

b) A .A. B

c) -C .P. I.LS.1

d) -I .A. J

e) (G .V. H .V. I).P.J - F .RS. J

f) (F .A. 77K) .LS. 6 .V. G .A. 77K .V. $ABCD$ .A. 7777 K4

2. Write a MAD main routine that will read in a positive integer value I, count the number of ones in its binary representation, print out I in octal form and print the count N in decimal integer form.

3. Write a MAD main routine that will read in two octal numbers R and S, and print them out along with their logical bitwise combinations: NR for "not" R, NS for "not" S, RAS for R "and" S, RVS for R "or" S and REVS for R "exclusive or" S.

4. Write a MAD main routine that will read in a character word named C and print out only the first three letters of that word.

5. Write a MAD main routine which will read in a character word named CHAR, put the first character into the last character position of C(1), the second character into the last character position of C(2), ..., and the sixth character into the last position of C(6). All other character positions in the elements of C should be blank. Then print out C(1) ... C(6).

6. Write a MAD main routine that will read in a character word named SHIFT, shift that word to the right so that any blanks on the right are eliminated and replaced with blanks on the left, and print the resulting word.

7. Write a MAD main routine that will read in a character word named QUEEN, determine whether that word contains the letter "Q", and print out a message of the form

```
THE WØRD
      QUEEN =      MYRTLE
DØES NØT CØNTAIN THE LETTER Q.
```

or a similar message if the word read in does contain the letter "Q".

8. Write a MAD main routine that will read in a character word named REVERS, reverse the order of the characters in that word and print out the result. For example, if the input word is EGRØEG, then the result which is output should be GEØRGE.

## Chapter Five

# DECISIONS AND TRANSFER OF CONTROL

Useful algorithms can seldom be written without providing for the making of decisions and thus allowing more than one path through the instructions. Even in the simple algorithm developed in Chapter One for the solution of the equation

$$ax + b = 0$$

there were three distinct paths, depending on the values of the data. Also in Chapter One, an algorithm designed to perform the same set of operations on more than one set of input data was presented. The transfer of control which was used in that program to get back to the input statement has many other applications as well.

Since any statement which is explicitly referenced elsewhere in a routine must be labeled, the form of these labels is presented at the beginning of this chapter. The unconditional transfer and the variable transfer or switch are then described. The switch involves a transfer of control to one of several labeled statements based on the value of an arithmetic expression.

Most of the decision-making features of the MAD language involve choosing paths dependent upon the truth value of logical or Boolean expressions. The most simple expressions of this type are the relations which were discussed briefly in Chapter One. The complete description of the structure and evaluation of relations and Boolean expressions is included in this chapter. Boolean assignment statements, which cause the assignment of truth values to variables, make it possible to work with such Boolean variables in a manner similar to that for arithmetic variables. The implementation in MAD of conditional statements, both simple and compound, which use these truth values as the basis for branching or alternate selection of paths, is then presented.

Some applications of these various features are illustrated by several more complex examples of algorithms which may now be introduced. In the last section, a complete program for solving quadratic equations is presented. Further examples involving the use of the features described in this chapter will occur throughout the remainder of the text.

## 5.1 Labeling of Statements

As was seen in Chapter Two, instructions in a computer are normally taken from successive memory locations during the execution of a program. The location of the next instruction to be executed is stored in the control register, which is incremented after each instruction. This normal flow of control can be modified by a transfer or jump instruction, which changes the contents of the control register.

In an algorithmic language such as MAD, statements are normally executed in the order in which they appear in the routine (or in the order in which the corresponding cards are located in the deck). If the order of execution of statements is to deviate from this sequential one, there must be some way to refer to the statement which is to be executed next. In MAD, any executable statement may bear a statement label and may thus be referenced.

A statement label consists of a name and possibly a constant subscript applied to that name. The appearance of a label on a statement causes that label to become a statement label variable. In the memory location reserved for that variable, the compiler automatically inserts a machine language transfer instruction to the labeled statement. Thus, the value of a statement label variable is a transfer instruction to a specific statement in the routine. Other variables of statement label mode may be used which do not label any statements.

Definition of a statement label name, label, and label variable:

```
<label-name>   is   <id>
    <label>    is   <label-name>  or  <label-name> ( <deci-con> )
<label-var>    is   <label-name>  or  <label-name> ( <a-exp> )
```

Statement labels may be punched anywhere in the first 10 columns of a card. They are thus restricted to a maximum of 10 characters, including parentheses and blanks. The subscript, if any, must be a non-negative decimal integer constant. For example, S, S8, START, END, END(16234) and ALWAYS(13) are legal statement labels: but S(-3), LABEL(I), ALWAYS(16234), TØØLØNG, B L A N K S and 36 are not. On the other hand, blanks are ignored in labels so that, for example, the labels TESTB and ANYX in Figure 1.4b could have been TEST B and ANY X, respectively. Since declarations, continuation cards of statements, and remarks may not be labeled, any characters punched in the first 10 columns of such cards are ignored by the compiler.

Any name that is used in a statement label (i.e. is used in the first 10 columns of an executable statement) is automatically declared to be in statement label mode. Thus, it may not be used as a variable name or a subroutine name or be declared to be in any other mode. Whenever the name of a statement label variable

does not appear in a statement label, the mode of that name must be declared using either

STATEMENT LABEL L{ <id> or { <id> or (L{<id>})} ( <structure> ) }

or

NØRMAL MØDE IS STATEMENT LABEL

In the latter case, the modes of all names other than those in statement label mode must be properly declared. Names appearing in labels may also be explicitly declared, but such redundant declarations are usually omitted.

As was implied by the appearance of a single subscript in the definitions above, statement label vectors are used in MAD. Since these vectors are stored in the same way as other types of vectors, space must be reserved in memory for all components up to and including the largest one to be used. The compiler automatically selects the largest subscript used with each name appearing in one or more subscripted labels and dimensions the vector accordingly. Therefore, unnecessarily large subscripts should not be used in statement labels. Statement label vectors may also be dimensioned with DIMENSIØN declarations or by including structure in a mode declaration, as indicated above.

There are no constants of statement label mode in MAD, but expressions of a simple nature do exist.

Definition of statement label mode expression:

<label-exp> is <label-var> or <label-fn-call>#

These expressions may be used to assign values (i.e. transfer instructions) to statement label variables using the statement label mode substitution statement which has the form:

<label-var> = <label-exp>

A statement may have only one label; it need not have any. In some cases, it is useful to have available a statement that causes no action to be performed, but serves simply as a labeling point. Examples of such situations will arise in section 5.3 where two or more labels are required for the same point in the routine and in Chapter Six where it is used as the termination of the iteration statement. The MAD statement for this purpose is

CØNTINUE

A card that contains a statement label, but no punches in columns 11 through 72, is treated as a CØNTINUE statement. Once again, the CØNTINUE statement causes no computation or operation, but may be labeled and transferred to.

## 5.2 The Unconditional Transfer

The most simple statement for the transfer of control to another (labeled) statement is the unconditional transfer statement which is of the form:[1]

TRANSFER TØ <label>

After the transfer statement, the next statement executed is the one specified by the value of the label contained in the transfer statement. This value is a transfer to the statement bearing that label, unless it has been changed (see discussion at the end of this section). Since the unconditional transfer statement always causes a transfer to the label, the statement following the TRANSFER TØ statement will never be executed unless it is labeled or may be reached in some other way. The existence of such statements (other than the END ØF PRØGRAM or END ØF FUNCTIØN statements) which cannot be reached is treated as a fatal compilation error in MAD and the following diagnostic is given:

***** ERRØR 33657 .... NØ WAY TØ REACH THIS STATEMENT.

One common use of the unconditional transfer is in a main routine that is designed to make the same calculations on several sets of data. An example of such a use is a routine designed to add sets of four numbers (each set to be read in as a data set) and print out the sum. The MAD statements for such a routine with some typical sets of data are shown in Figure 5.1a and the flowchart for this routine is in Figure 5.1b. This algorithm is typical of a large class of algorithms, the flowcharts for which are of the form of Figure 5.1c. Note that the unconditional transfer statement requires no box in a flowchart. It is usually represented only by a directed line, often terminating in a connector box, a circle (as in these flowcharts). The output resulting from this program would be as follows:

```
A=3.6,4.1,-9.8,7.6*
       SUM =  5.500000
A(0)=1.,2.,3.,4.*
       SUM = 10.000000
A(0)=-2.9,A(1)=-9.6,A(3)=15.1,A(2)=-6.*
       SUM = -3.400000
****ALL INPUT DATA HAVE BEEN PRØCESSED
```

[1] The variable transfer or switch, section 5.3, is a different type of transfer statement, also using the same words, TRANSFER TØ.

```
          RRØUTINE TØ ADD FØUR NUMBERS
           DIMENSIØN A(3)
START      READ AND PRINT DATA
           SUM = A + A(1) + A(2) + A(3)
           PRINT RESULTS SUM
           TRANSFER TØ START
           END ØF PRØGRAM
$DATA
A=3.6,4.1,-9.8,7.6*
A(0)=1.,2.,3.,4.*
A(0)=-2.9,A(1)=-9.6,A(3)=15.1,A(2)=-6.*
```

Figure 5.1a

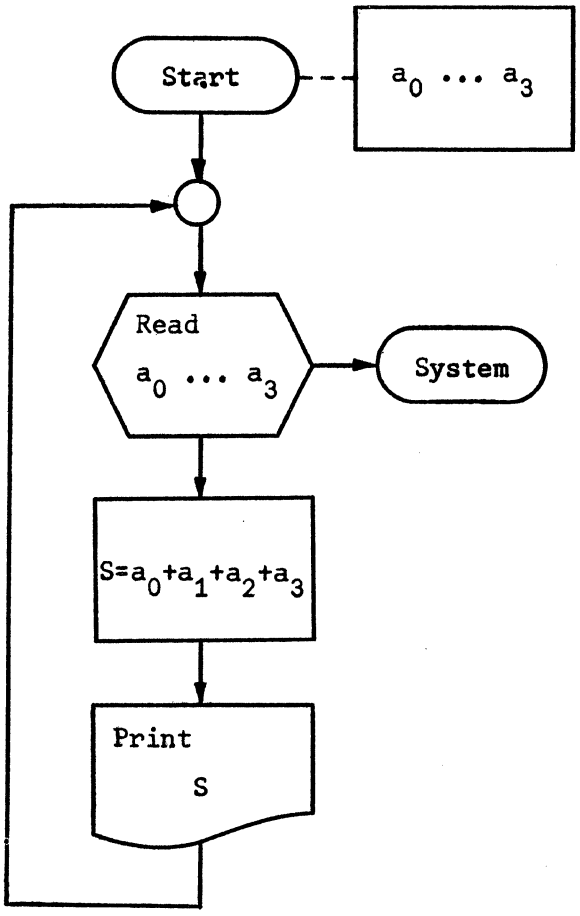



Figure 5.1b

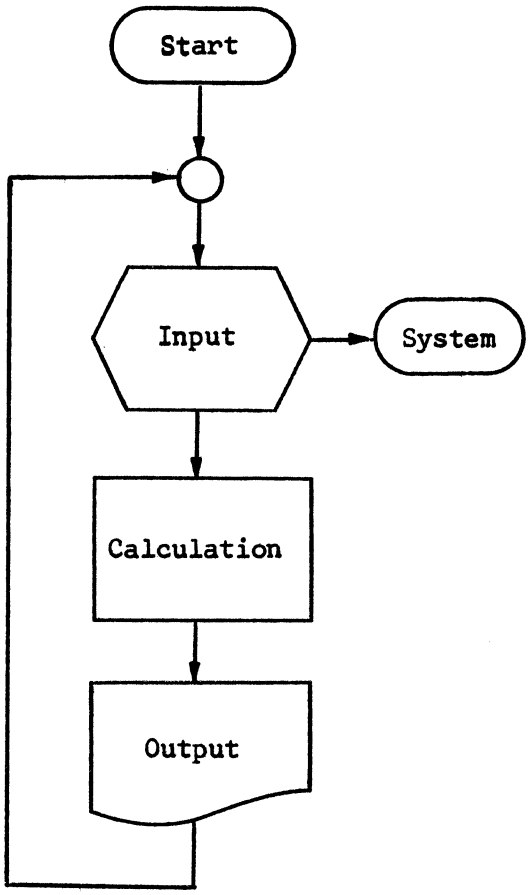



Figure 5.1c

An unconditional transfer of control is often used to cause a sequence of instructions to be executed repeatedly as in the previous example. In that case, the loop was terminated when the input statement did not find any further data sets. If there are no conditions under which control can pass outside this sequence, then the computer becomes trapped in an "infinite loop". Although the computer continues to go in non-productive circles in this case, the program is usually terminated after some fixed amount of time has elapsed, as a result of excessive output, or for some other reason. For example, the following MAD routine would cause the computer to become trapped in such a loop.

```
      X = 0.0
LØØP  PRINT RESULTS X
      X + X + 1.0
      TRANSFER TØ LØØP
      END ØF PRØGRAM
```

Since this sequence contains a print statement, an excessive number of pages of output would be produced. In MAD such a routine would be terminated because of this excessive output and an execution time diagnostic of the following form printed:

```
****PAGE CØUNT EXCEEDED
    AT LØCATIØN 10342
```

As a further, somewhat ridiculous, example the following routine contains a very tight loop!

```
SUNK  TRANSFER TØ SUNK
      END ØF PRØGRAM
```

Such a loop which contains no output statements will normally be terminated only by exceeding a time estimate built into the system or specified by the user. In this case, the execution time diagnostic would be of the form:

```
****TIME ESTIMATE EXCEEDED
    AT LØCATIØN 10462
```

In some more unusual cases, the machine may become caught in a loop that destroys the systems programs, and thus the computer system may be incapable of detecting such errors. The machine operator must then intervene and put the machine back onto a productive course.

The value of a statement label variable may be changed by a substitution statement. For example, if START and END are statement labels, the statement

```
START = END
```

would cause a TRANSFER TØ START statement to behave like TRANSFER TØ END. After execution of this statement, both of these statement label variables would contain a transfer instruction to the statement labeled END and the transfer instruction to the statement labeled START, originally stored in the variable START, is destroyed.

Exercises

1. Which of the following are legal MAD statement labels? If not, why not?

| | | | | | |
|---|---|---|---|---|---|
| a) | AB | g) | INDEX(23) | m) | B(I*J+2) |
| b) | 31 | h) | INDEX( 213 ) | n) | XY(-26) |
| c) | A2641 | i) | 3E27 | o) | A(+3) |
| d) | X(2137) | j) | STATEMENT | p) | GØ TØ |
| e) | INDEX(2137) | k) | LABEL | q) | STRAT |
| f) | INDEX(I) | l) | Z(l+3) | r) | A(3.6) |

2. Which of the following are correct unconditional transfer statements in MAD? If not, why not?

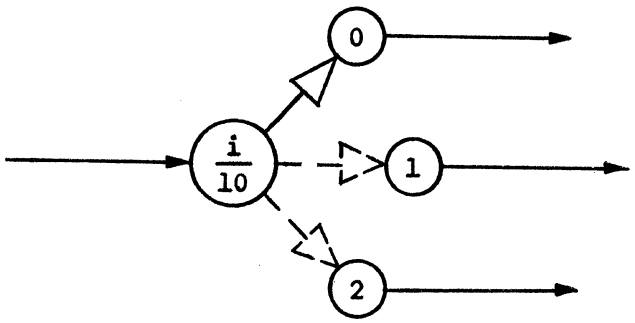
| | | | |
|---|---|---|---|
| a) | TRANSFER TØ MAD | f) | TRANSFER TØ L(I) |
| b) | TRANSFER TØ 735 | g) | TRANSFER TØ L(L) |
| c) | TRANSFER TØ MARYLAND | h) | TRANSFER TØ NEXT |
| d) | TRANSFER TØ L(23) | i) | TRANSFER TØ (13) |
| e) | TRANSFER TØ L23 | j) | TRANSFER TØ L(I+3) |

3. Describe and correct any errors in the following MAD routine.

```
RPRØGRAM FØR THE ADDITIØN ØF THREE
1NUMBERS A(0), A(1), A(2)
 READ AND PRINT DATA        A(0) ... A(2)
 SUM = A(0) + A(1) + A(2)
 TRANSFER TØ A
 PRINT RESULT SUM
 END ØF PRØGRAM
 DIMENSIØN A(2)
```

## 5.3 The Variable Transfer or Switch

It is often desirable to be able to transfer to one of several places in a routine, depending on the value of a variable or a more general arithmetic expression. Such a transfer might appear in a flowchart as follows:



where the action appears as that of a switch, the connection depending on the value of the expression. (Although three connections are shown in this example, any number is possible.) When the value of i/10 is 0, the switch is set to the zero position as shown above, and the steps following that circle are executed next.

To do this in MAD, the various statements to which control may be transferred are given labels with a common name, but different constant subscripts. The general form of the variable transfer or switch statement in MAD is:

TRANSFER TØ <label-name> ( <a-exp> )

The value of the arithmetic expression (converted to integer mode, if necessary) determines the element of the statement label vector to which control is transferred and thus the statement which is executed next. If that element of the vector has not been given a value (e.g. the value of the expression is not one of those expected) the execution of the program is usually terminated with a diagnostic such as

```
**** WHAT WERE YØU DØING
     AT LØCATIØN 10136
```

being printed, where the location specified is the address in the statement label vector to which the computer transferred. However, if the subscripted element is not within the statement label vector, the machine will sometimes succeed in making a transfer and continue executing instructions; completely meaningless results will usually be produced after such an occurrence.

The switch in the flowchart above could be realized with a statement of the form

TRANSFER TØ S(I/10)

where the labels S(0) (or S), S(1), and S(2) appear somewhere in the routine.

One of the most common applications of such a transfer is in a coding or decoding problem, where the purpose of the routine is, for example, to assign a code word to each input word.

An example of a problem in this class is the following:

> Given, an integer numeric grade, g, as input data (where $0 \leq g \leq 100$), calculate and print out the corresponding letter grade, using the following scale:
>
> | | |
> |---|---|
> | 90-100 | A |
> | 80-89 | B |
> | 70-79 | C |
> | 60-69 | D |
> | 0-59 | F |

Let us now consider some algorithms for the solution of this problem. A straightforward approach would involve a 101-way switch, transferring to a label having the numerical grade as its subscript. Although individual statements could be used to print the letter grade corresponding to each of the 101 possibilities, a more efficient algorithm would use CØNTINUE statements to place all of the labels corresponding to a letter grade on the same print statement. Either of these algorithms would be executed efficiently; however, the corresponding MAD programs would be quite lengthy. Much time would be consumed in writing, keypunching, and flowcharting such a program. With some thought, much shorter algorithms can be written which will compile and load faster, and occupy less memory space.

If we realize, however, that the second digit of the numeric grade has no bearing on the letter grade (e.g. all grades between 70 and 79, inclusive, correspond to a letter grade of C, the complexity of the switch can be reduced by making the switch setting depend on g/10 (integer mode). This would now require an 11 position switch, since values of g/10 range from 0 through 10. A flowchart for such an arrangement is shown in Figure 5.2a, and the corresponding MAD routine in Figure 5.2b. This approach still results in a fairly complicated flowchart and several wasted labels. Let us, thus, consider a two-level decision process, i.e. one utilizing two variable transfers. First, let us divide the grades into two groups, namely passing and failing. This can most easily be done by using the integer mode quotient g/60 to control a switch. The switch is set to one whenever $g \geq 60$, i.e. the grade is passing, and to zero otherwise. We can now classify the passing grades by utilizing a switch with control $\frac{g - 60}{10}$. The value of that expression will range from 0 (when g is between 60 and 69) to 4 (when g is 100). A flowchart for that routine is shown in Figure 5.3a and the corresponding MAD routine in Figure 5.3b.

A few comments on the routines of Figures 5.2 and 5.3 are in order. First, since the first 6 elements of the label vector in Figure 5.2b are intended for the same statement, namely, PRINT CØMMENT $ F$, five CØNTINUE statements are required. Second, the elements of a statement label vector need not be used in numerical order (Figure 5.3b). Either S or S(0) may be used to denote that element of the vector. Finally, the comments could all be modified to the form:

```
PRINT CØMMENT $+                    LETTER GRADE IS X.$
```

(with the appropriate letter grade X.). This will allow the printing of that statement on the same line with the data (without overprinting if the data is confined to the first 20 columns of the card).

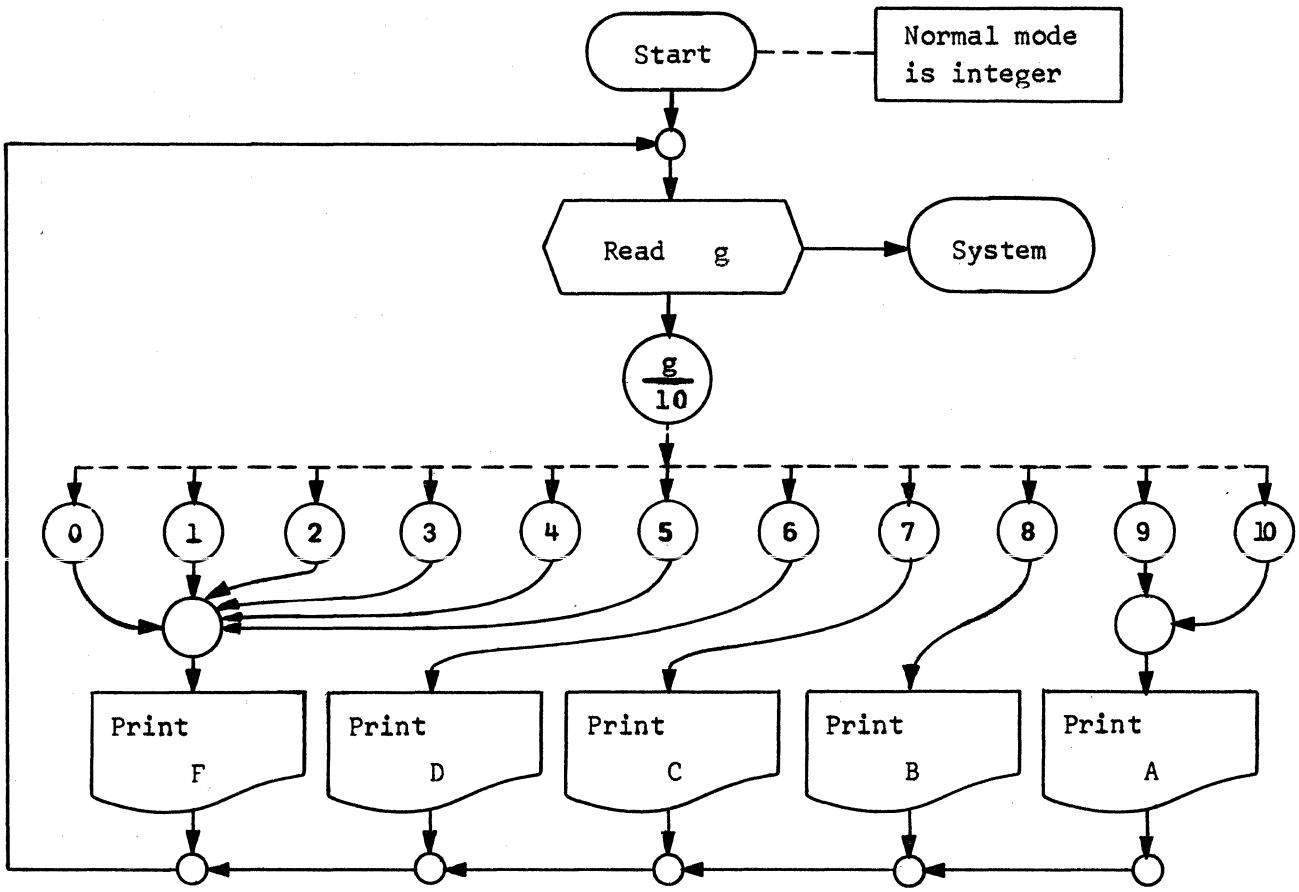



Figure 5.2a

```
          RRØUTINE TØ CØNVERT NUMERICAL GRADES TØ LETTER GRADES
           NØRMAL MØDE IS INTEGER
START     READ AND PRINT DATA       G
          TRANSFER TØ S(G/10)
S(0)      CØNTINUE
S(1)      CØNTINUE
S(2)      CØNTINUE
S(3)      CØNTINUE
S(4)      CØNTINUE
S(5)      PRINT CØMMENT $ F$
          TRANSFER TØ START
S(6)      PRINT CØMMENT $ D$
          TRANSFER TØ START
S(7)      PRINT CØMMENT $ C$
          TRANSFER TØ START
S(8)      PRINT CØMMENT $ B$
          TRANSFER TØ START
S(9)      CØNTINUE
S(10)     PRINT CØMMENT $ A$
          TRANSFER TØ START
          END ØF PRØGRAM
```

Figure 5.2b

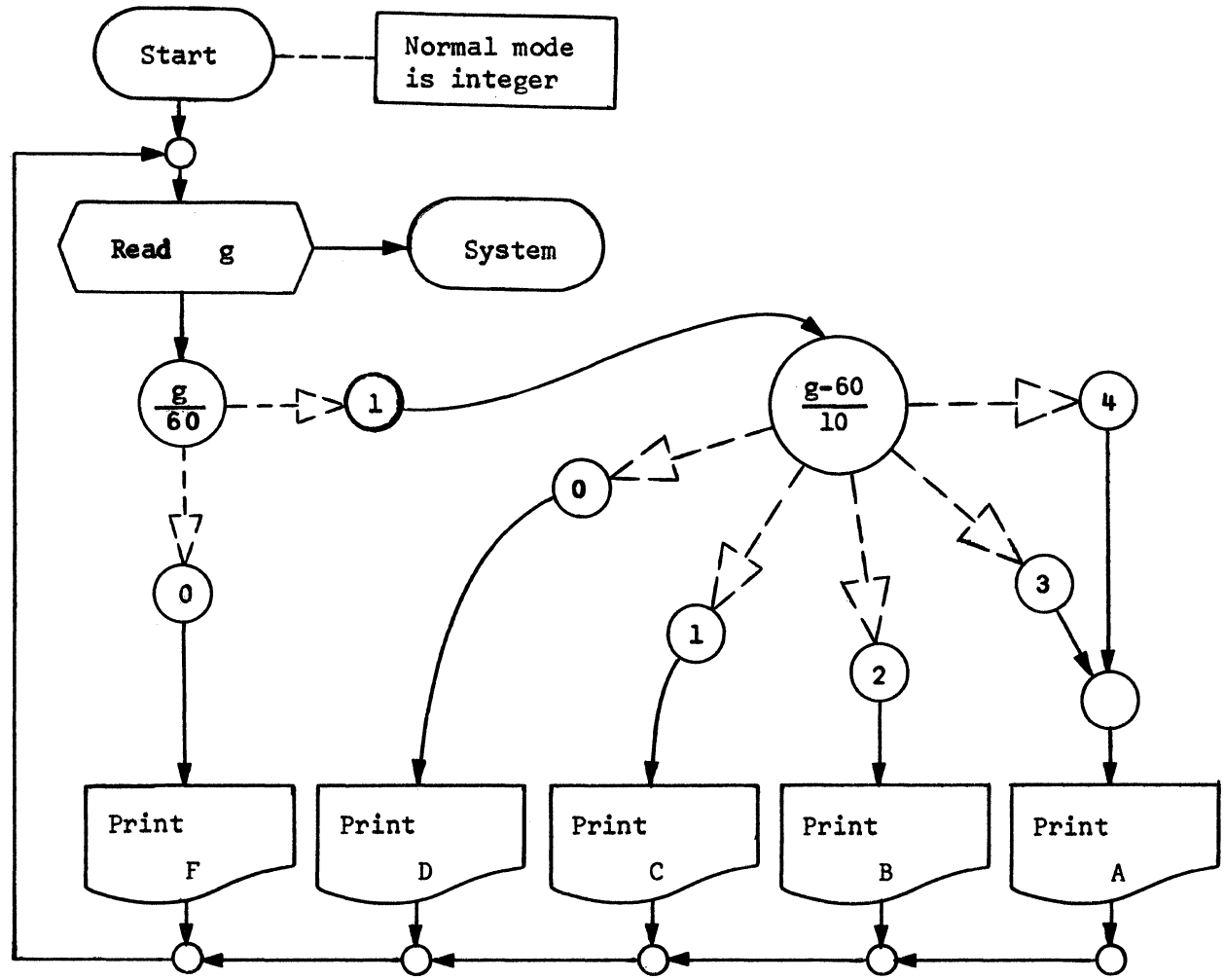



Figure 5.3a

```
          RRØUTINE TØ CØNVERT NUMERICAL GRADES TØ LETTER GRADES
           NØRMAL MØDE IS INTEGER
START      READ AND PRINT DATA       G
           TRANSFER TØ T(G/60)
T(0)       PRINT CØMMENT $ F$
           TRANSFER TØ START
T(1)       TRANSFER TØ S((G-60)/10)
S(3)       CØNTINUE
S(4)       PRINT CØMMENT $ A$
           TRANSFER TØ START
S(2)       PRINT CØMMENT $ B$
           TRANSFER TØ START
S(1)       PRINT CØMMENT $ C$
           TRANSFER TØ START
S          PRINT CØMMENT $ D$
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 5.3b

Exercises

1. Draw a flowchart and write a MAD routine to convert a decimal integer N (where $0 \leq N \leq 7$) into a three bit binary number. The data is N and the result, BINARY, is to be printed (e.g., if N=6, then BINARY = 110).

2. A three-way switch, the position of which depends on whether an arithmetic expression is negative, zero, or positive, is available in some other languages. How can this be realized using the variable transfer? Use this switch in a MAD main routine to print out an appropriate comment about the sign of an input variable X. (Hint: In MAD, if X is zero, then .ABS.X/X is always zero.)

## 5.4 Boolean Mode and Relations

In Chapter 4, two computational modes--integer and floating point--were discussed. In each mode, constants, variables, operators and expressions were defined. An important feature of elements of these modes is their interchangeability, i.e. the ability to mix integer mode and floating point mode elements in many expressions. A third mode, statement label mode, was discussed in detail earlier in this chapter. However, operators and expressions are not defined for statement labels.

In this section, we will describe in detail a fourth mode, Boolean mode, in which operations may be performed and assignments of value made with a structure nearly parallel to that of floating point and integer modes. Within Boolean mode, constants, variables, operators and expressions are defined; but, these may not be mixed with the corresponding elements in other modes. In addition to these parts, relations and relational operators are defined which produce Boolean mode results from arithmetic operands. The values of Boolean expressions are the basis for many branches and the termination of most iterations in MAD.

As discussed in Chapter 3, there are two constants in Boolean mode - "true" (which is written and printed as 1B in MAD), and "false" (written as 0B). The value of all Boolean mode variables, relations, and expressions is thus either true or false.

Boolean variables, <bool-var>, are constructed following the same rules as for other variables. The mode of Boolean variables must be declared, either using

```
BØØLEAN L{ <bool-var> }
```

or by making Boolean the normal mode, using

```
NØRMAL MØDE IS BØØLEAN
```

Note that in the latter case, the mode of all variables not in Boolean mode must be declared. Boolean mode vectors and arrays may be used and they follow the same subscripting and dimensioning rules as other vectors and arrays. (See section 3.6.) Note that subscripts are integer or floating point mode expressions, not Boolean mode expressions.

In many applications, the only Boolean expressions that are used are relations. We will defer the complete definition of Boolean expressions until after defining relations and discussing their use in the simple conditional statements.

In the strict mathematical sense, a relation, as used in MAD, is not an operation. It is a mapping of the set of ordered pairs of numbers into another set, namely, the set of two Boolean values. Although the two numbers must be of the same mode when the relation is evaluated, integer mode numbers will automatically be converted to floating point if necessary. The six relational operators are all binary, each representing a comparison between two arithmetic expressions. The operations and the equivalent algebraic and MAD notation are shown in Table 5.1. It should be

Table 5.1 Relational Operators

| Operation | Algebraic Operator | MAD Operator |
|---|---|---|
| greater than | $>$ | .G. |
| greater than or equal to | $\geq$ | .GE. |
| less than | $<$ | .L. |
| less than or equal to | $\leq$ | .LE. |
| equal to | $=$ | .E. |
| not equal to | $\neq$ | .NE. |

emphasized that the relational operator "equal to" is always written .E., never with an equal sign (=). This is one of the most common errors in MAD and a difficult one to find because the equal sign looks so right. Formally we now give the Definition of relational operator and relation:

<rel-op> is .G. or .GE. or .L. or .LE. or .E. or .NE.

<rel> is <a-exp> <rel-op> <a-exp>

A relation is evaluated as follows:

1) Each arithmetic expression is evaluated independently (according to the rules discussed in Chapter 4).
2) The expression on the right of the operator is subtracted from that on the left (with conversion of either operand from integer mode to floating point mode if necessary to make them agree in mode).
3) The result of step 2 is compared to zero and the value of the relation (either <u>true</u> or <u>false</u>) is determined.

Some examples of relations are:

```
          3.6  .G.  1.9
            I  .LE.  J-1
2.*P - 3..P.2  .E.  4.6/(2.* X) + 3.9
            2  .NE.  2
```

Although there is nothing actually incorrect about relations (such as the first and last ones above) whose values are constant, they should be replaced by their constant values in order to avoid wasting computer time when the routine is executed. Another important point is that the operator .E. means <u>exactly</u> equal to, not approximately equal to. This is particularly important when the operation is on floating point mode numbers, where small roundoff errors introduced in conversion from decimal to binary or in arithmetic operations can cause slight differences in values. For example, the value of the relation

```
6.3 - 6.  .NE.  .3
```

is <u>true</u> since the expression 6.3-6.-.3 differs from zero in the last few bits. The reader should verify this example as an exercise. Some further examples of relations, shown after the variables involved have been replaced by their arithmetic values, and the values of these relations are:

| Boolean | Value |
|---|---|
| 5 .G. 6 | False |
| 5 .NE. 6 | True |
| 3.14 .E. 3.14159 | False |
| 3.15 .LE. 3.15 | True |
| (9.P..5) .GE. 3. | True |
| 1 .L. -2 | False |

The relational operators are all equal in precedence, but are lower on the precedence list than any of the arithmetic operators. Thus, all arithmetic expressions are evaluated before the relation is tested. Therefore, the parentheses used in one of the examples above are unnecessary.

Exercises

1. If PI = 3.14, R = 2.0, AREA = 12.57, and D = 4.0, indicate which of the following relations have the value true and which false, and why:

a) AREA .E. PI*R.P.2

b) AREA .G. D * PI

c) D .NE. 2*R

d) AREA - PI*R*R -1E-2 .E. 0

e) 3/2*2 .E. 3

f) .ABS. - R.P.2 .G. D

g) J.P.2 .GE. 0

h) .ABS.K .LE. 0

2. For each of these pairs of expressions, state whether the value of the relation A.GE.B is true or false.

| | A | B |
|---|---|---|
| a) | 5E6 | 6E5 |
| b) | .0036 | 4.2E-4 |
| c) | -14E1 | -270 |
| d) | 7.2E2 | 720 |
| e) | 1E8 | 97031010 |
| f) | -5.4697 | 3.1231 |
| g) | .ABS. (22) | 22 |
| h) | .ABS. (-22) | -22 |
| i) | $A$ | $B$ |
| j) | $A$ | 21K10 |

## 5.5 The Simple Conditional

The simple conditional statement often provides the most convenient means of causing decisions to be made in MAD routines. This statement uses the value of a Boolean expression to determine whether or not a simple statement is to be executed. These simple statements, which will be described in this section, are actually part of the simple conditional statement. For the case when the simple statement involved is an unconditional transfer, the simple conditional provides for a two-way branch dependent on the value of the Boolean expression. The more versatile compound conditional will be discussed in section 5.7.

The simple conditional statement consists of a conditional clause, containing the Boolean expression, followed by the simple statement which is to be controlled. Whenever the value of the Boolean expression is true, the simple statement is executed; otherwise, its execution is omitted.

The general form of the simple conditional statement is:

WHENEVER <bool-exp>, <simple-statement>

where we use the

Definitions of a Boolean expression and a simple statement:

| | | |
|---|---|---|
| <bool-exp> | is | <bool-con> or <bool-var> or <rel><br>or some more complex Boolean expression<br>as defined in section 5.6. |
| <simple-statement> | is | any executable statement (not a<br>declaration) except END ØF PRØGRAM,<br>simple and compound conditional#<br>statements, iteration statements#,<br>or the ENTRY TØ# statement which<br>is used in subroutines. |

In the remainder of this section, we will use only Boolean constants, Boolean variables and relations as Boolean expressions in the examples; more complex examples will be discussed in the next section. Some examples of simple conditional statements are:

```
WHENEVER B .E. 0, PRINT CØMMENT $ YØU DID IT AGAIN.$
WHENEVER A(J) .GE. 10.3 - 4.*X, SUM = SUM + A(J)
WHENEVER P, TRANSFER TØ GØ
```

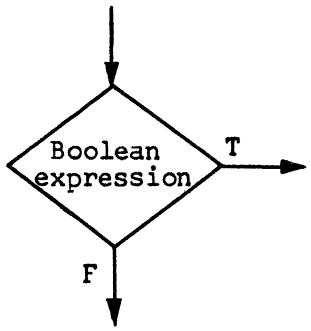
where P is a Boolean mode variable and all other variables are arithmetic.

The operation of the simple conditional can best be described if two separate

cases are considered. The first case is where the simple statement is a transfer statement, in which case the sequence of events is as follows:

1) The Boolean expression is evaluated.
2) If its value is true, the transfer is executed and execution continues at that point.
3) Otherwise, the next statement in the routine is executed. (Except if the conditional is the last statement in a statement set of a compound conditional# or the last statement in the scope of an iteration statement#.

This statement may be represented in a flowchart by a decision box such as




where F indicates the path to be followed when the value of the Boolean expression is false and T indicates the path when the value is true. When the Boolean expression is simply a Boolean constant, the statement behaves like an unconditional transfer or a CØNTINUE statement depending on whether the constant is true or false, respectively. Computing time, however, is wasted making a useless check in these cases. The algorithms for the two problems discussed in Chapter One involve branches of this type. A further example of such a situation is the following problem:

> Floating point mode numbers are to be read in one at a time. When the number is positive or zero, the positive square root is to be printed. Otherwise, a comment is to be printed that the root is imaginary and then the magnitude of the root is to be printed.

A flow chart of an algorithm and the corresponding MAD coding for the solution of this problem are shown in Figure 5.4. If, for example, the two data sets are

```
X = 4.*
X = -16.*
```

the order of execution of this program is as follows:

> The first set of data (X = 4.) is read and printed out. The relation X.GE.0. is evaluated. Since it is true for this case, i.e. 4. is greater than or equal to zero, the next statement executed is NONNEG and Y is set equal to 2.0. The result is printed and a new set of data, namely X = -16., is read. The relation is evaluated and found to be false for this case. Thus, the next statement, X = -X is executed and X is made equal to +16.0. The comment is then printed; Y is calculated to be 4.0; and the result is

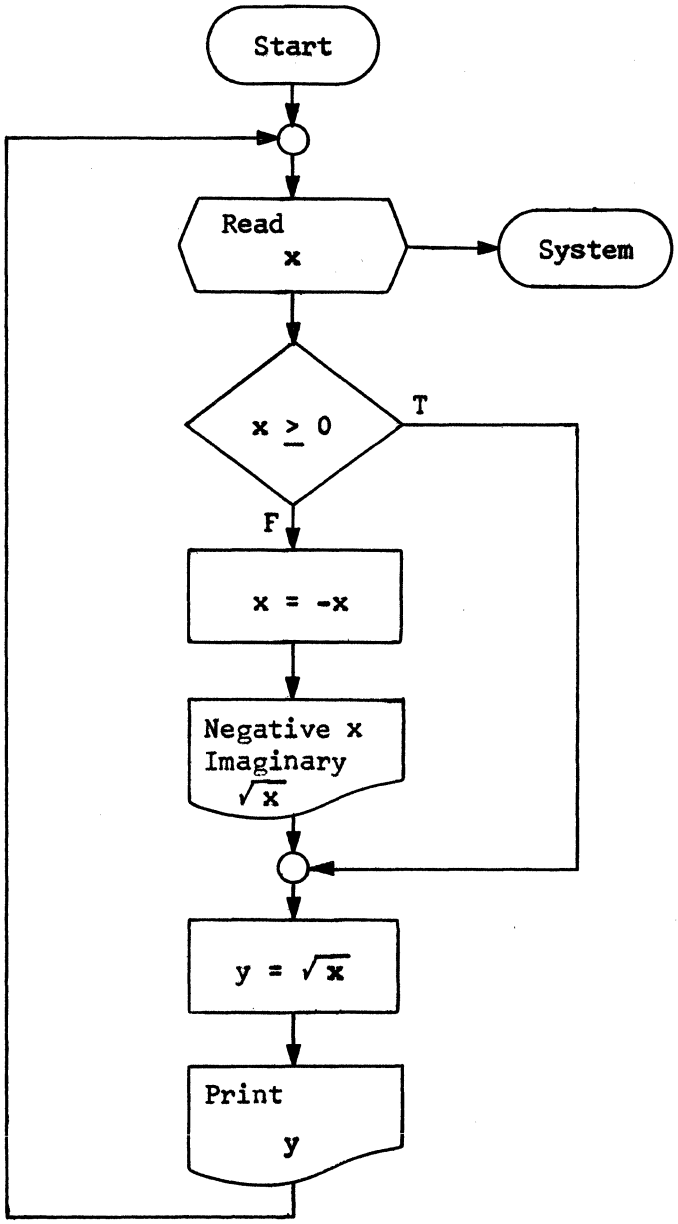



Figure 5.4a

```
          REXAMPLE ØF A SIMPLE CØNDITIØNAL BRANCH
START     READ AND PRINT DATA              X
          WHENEVER X .GE. 0., TRANSFER TØ NØNNEG
          X = -X
          PRINT CØMMENT $ X IS NEGATIVE. THE SQUARE RØØT IS IMAGINARY.$
NØNNEG    Y = X.P..5
          PRINT RESULTS Y
          TRANSFER TØ START
          END ØF PRØGRAM
```

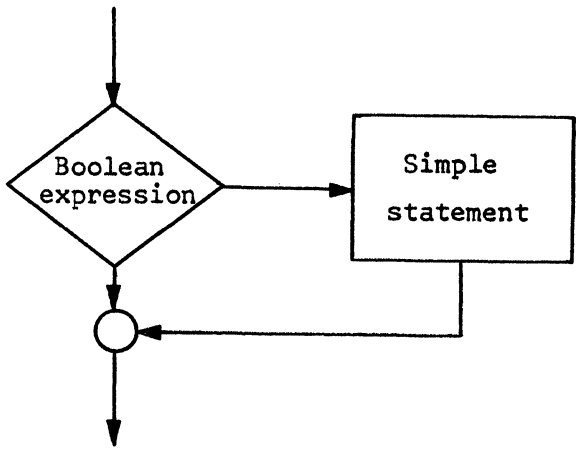
Figure 5.4b

then printed. The transfer is executed and an attempt is made to read a new set of data. Since there are no more data sets, the program is completed and control is returned to the system.

The second case of the simple conditional is where the simple statement is not a transfer statement. Then, the sequence of events is as follows:

1) The Boolean expression is evaluated.
2) If its value is true, then the simple statement is executed.
3) In either case, the next statement in the routine is executed (except if the simple conditional is the last statement in a statement set of a compound conditional# or the last statement in the scope of an iteration statement#).

The statement would be represented in a flowchart by the boxes




The solution of the following problem serves as an illustration of an algorithm where this second form of the statement might be useful.

Sets of three floating point mode numbers, $a_0$, $a_1$, and $a_2$, are to be read as data. Calculate and print out the sum of the negative numbers from each set. A flowchart of an algorithm for the solution of the problem and the corresponding MAD coding are shown in Figure 5.5a and 5.5b. The printout resulting from some typical sets of data is shown in Figure 5.5c.

Another application of the branching form of the simple conditional statement is in a routine designed to perform two or more different functions, depending on the value of a flag. A flag is a variable, the value of which has no numerical significance, but which is used only to control the position of a switch or to determine which branches are taken in the routine. The flag may be an integer mode or floating point mode variable, and, when only two functions are required, the flag is often a Boolean mode variable. With the language tools available at this point, we can give only a simple example of the use of a flag. The power of the flag will become more clear in later chapters.

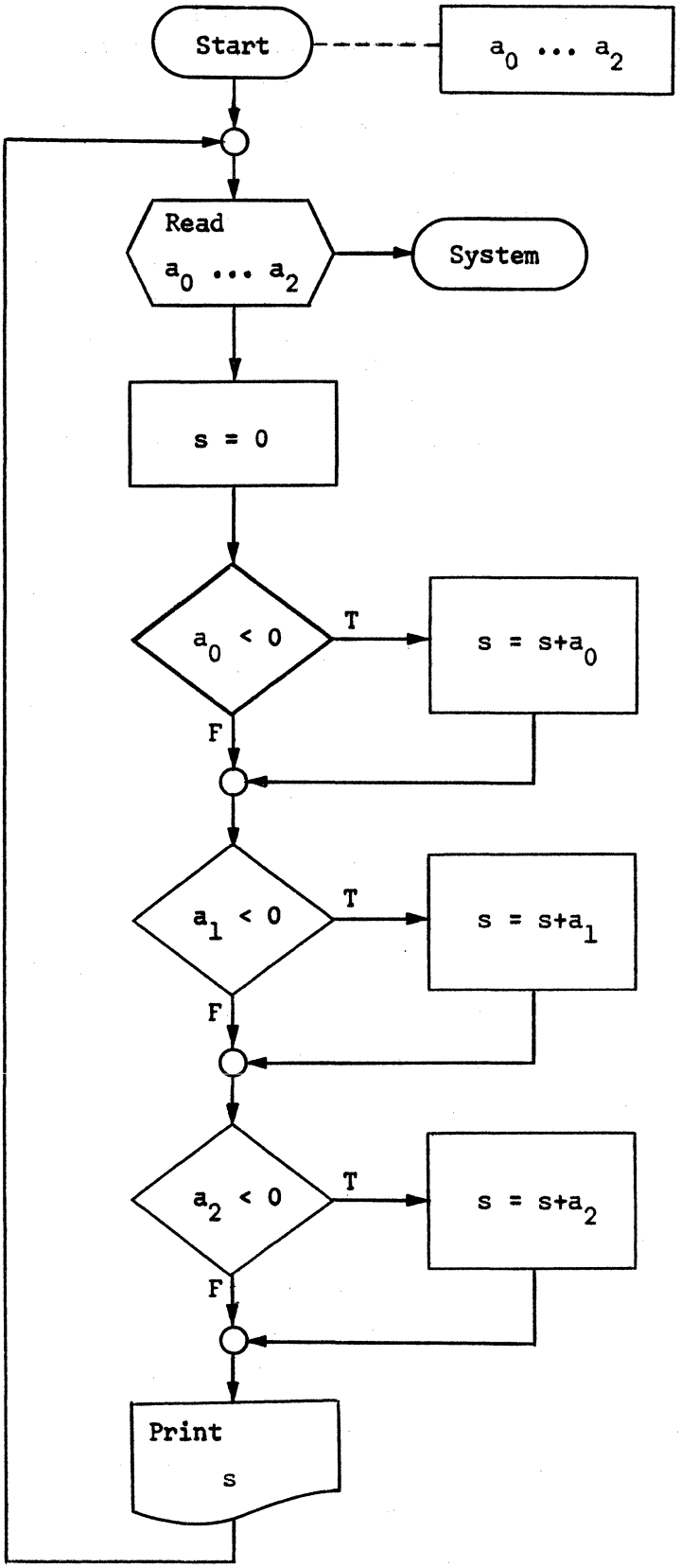

Start
$a_0 \ldots a_2$
Read
$a_0 \ldots a_2$
System
s = 0
$a_0 < 0$
T
$s = s+a_0$
F
$a_1 < 0$
T
$s = s+a_1$
F
$a_2 < 0$
T
$s = s+a_2$
F
Print
s

Figure 5.5a

```
          RPRØGRAM TØ FIND THE SUM ØF THE NEGATIVE NUMBERS
          RØF A SET ØF THREE NUMBERS.
           DIMENSIØN A(2)
START      READ AND PRINT DATA        A(0)...A(2)
           SUM = 0.
           WHENEVER A(0).L.0., SUM = SUM + A(0)
           WHENEVER A(1).L.0., SUM = SUM + A(1)
           WHENEVER A(2).L.0., SUM = SUM + A(2)
           PRINT RESULTS SUM
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 5.5b

```
A(0) = 4.6, 9.1, 0.0 *

       SUM =  0.000000

A(0) = -6.2, 2.1, 4.1 *

       SUM = -6.200000

A(0) = -1., -2., -3. *

       SUM = -6.000000

****ALL INPUT DATA HAVE BEEN PRØCESSED
```

Figure 5.5c

As an example of the use of a flag, consider the following problem:

> Sets of 5 floating point numbers, $a_0 \ldots a_4$, are to be read as data, one at a time. Based on the value of a flag, either the arithmetic mean (average) or the magnitude of the arithmetic mean of those numbers is to be calculated.

We did not specify the value of the flag in the problem statement in order to illustrate the use of both Boolean and numeric flags. The algorithm consists of the input of data, calculation of the mean, calculation of the magnitude of the mean if that is desired, and the printing of results.

If an integer mode flag is used, most commonly the integers 0 and 1 are used

for the two values. In that case, the branch can be realized either using the variable transfer or, as is more common, the simple conditional statement. The Boolean expression in the conditional would then normally be FLAG .E. 0 or FLAG .E. 1. If a floating point mode flag were used, the values usually chosen are 0. and 1., or -1. and +1., with typical Boolean expressions being FLAG .G. .5 in the first case or FLAG .G. 0. in the second. If a Boolean mode flag is used, the values are, of course 0B and 1B. A flowchart of an algorithm using an integer mode flag, with value 0 indicating the mean and value 1 indicating the magnitude of the mean is shown in Figure 5.6a. A comparable routine utilizing a Boolean mode flag is shown in Figure 5.6b. In that routine the value <u>false</u> is used to get the mean and <u>true</u> to get the magnitude of the mean.

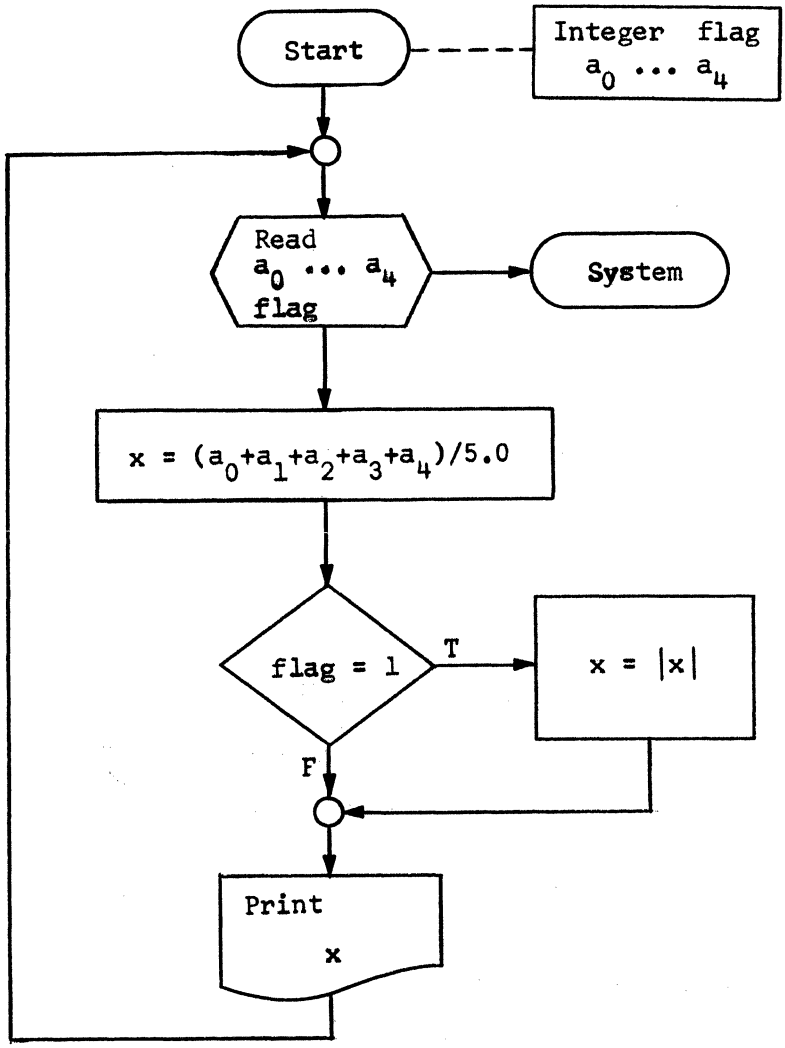



Figure 5.6a

```
          RRØUTINE TØ CALCULATE THE MEAN ØF A(0)...A(4) WHEN FLAG = 0B
          RAND THE MAGNITUDE ØF THE MEAN WHEN FLAG = 1B.
           BØØLEAN FLAG
           DIMENSIØN A(4)
START      READ AND PRINT DATA        A(0)...A(4)
           ANSWER = (A + A(1) + A(2) + A(3) + A(4)) / 5.
           WHENEVER FLAG, ANSWER = .ABS. ANSWER
           PRINT RESULTS ANSWER
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 5.6b

Exercises

1. For the following MAD program:

   a) Draw a flowchart.

   b) Explain (in words) what happens for various sets of input data. Be sure to include all special cases.

   c) List what would be printed out for a sufficient number of sets of data to illustrate all cases in b).

```
START      READ AND PRINT DATA     A, B, AND C
           WHENEVER B*B-4.*A*C .E. 0., PRINT CØMMENT $ TWØ EQUAL RØØTS.$
NEXT       WHENEVER A .E. 1., TRANSFER TØ NEXT
           WHENEVER A.P..5 .E. 0, A = A + 1
           WHENEVER 1/2*A .G. 0, PRINT CØMMENT $ ØH ØH $
           PRINT RESULTS A
           TRANSFER TØ START
           END ØF PRØGRAM
```

2. Write a MAD statement or sequence of MAD statements to do each of the following:

   a) If A (an integer) is odd, add one to it. Otherwise, do nothing.

   b) If B is positive, calculate its negative square root. Otherwise transfer to a statement labeled GØØF.

   c) Print a 1 (and nothing else) if X is greater than 10 and print a 2 (and nothing else) if X is less than or equal to 10.

3. Write a MAD routine for the algorithm described by the flowchart in Figure 5.6a using an integer variable flag.

4. Draw a flowchart and write a MAD routine to calculate and print out the sum of 6 numbers, $a_1 \ldots a_6$. The result should be printed in decimal if the value of the flag is 0 and in octal if it is 1. ($a_1 \ldots a_6$ and flag are the inputs).

## 5.6 Boolean Expressions and the Boolean Substitution Statement

In order to define the structure of Boolean expressions that are more complex than just constants, variables or relations, we will first introduce the six Boolean operators. Table 5.2 lists the operators with typical logical, algebraic and MAD notation, and Table 5.3 shows the truth tables for these operators.

Table 5.2 Boolean Operations

| Logical Operation | Algebraic Notation | MAD Notation |
|---|---|---|
| negation | $\sim a$ | .NØT. A |
| disjunction (inclusive) | $a \vee b$ | A .ØR. B |
| conjunction | $a \wedge b$ | A .AND. B |
| disjunction (exclusive) | $a \oplus b$ | A .EXØR. B |
| equivalence | $a \equiv b$ | A .EQV. B |
| implication | $a \rightarrow b$ | A .THEN. B |

Table 5.3 Boolean Operation Truth Tables

| A | B | .NØT. A | A .ØR. B | A .AND. B |
|---|---|---|---|---|
| false | false | true | false | false |
| false | true | true | true | false |
| true | false | false | true | false |
| true | true | false | true | true |

| A | B | A .EXØR. B | A .EQV. B | A .THEN. B |
|---|---|---|---|---|
| false | false | false | true | true |
| false | true | true | false | true |
| true | false | true | false | false |
| true | true | false | true | true |

The truth tables can be interpreted as follows: The operator being used determines the column of the table to be used. The value of the operand for .NØT. or operands for the other operators determine the row of the truth table. For example, if the value of the first operand is false and the value of the second operand is true, the second row is used. Thus, 0B .AND. 1B has value 0B, and 0B .EXØR. 1B has value 1B. The first three operators, .NØT., ØR., and .AND. are the most commonly used. The values produced by these operators may be stated in words as:

| | | |
|---|---|---|
| .NØT. A | is | true if A is false and is false if A is true. |
| A .ØR. B | is | true if either A or B or both are true, and is false only if both A and B are false. |
| A .AND. B | is | true if both A and B are true, and is false if either A or B or both are false. |

Formalizing the definitions of these Boolean operators, the definition of a Boolean expression follows easily along the same lines as that of an arithmetic expression.

Definition of unary and binary Boolean operators, and Boolean expressions:

| | | |
|---|---|---|
| <un-bool-op> | is | .NØT. |
| <bin-bool-op> | is | .ØR. or .AND. or .EXØR. or .EQV. or .THEN. |
| <bool-exp> | is | <bool-con> or <bool-var> or <rel> |
| | | or <bool-fn-call># or ( <bool-exp> ) |
| | | or <un-bool-op> <bool-exp> |
| | | or <bool-exp> <bin-bool-op> <bool-exp> |

Some examples of Boolean expressions are:

1B

A

X .G. 5.3

(X .E. Y)

.NØT. A

(Y.G.3) .ØR. B

(A .AND. .NØT. B) .ØR. ( (.NØT. A) .AND. B )

(X.P.2 - .ABS.(X*Y)) .G. (Y/X)

where A and B are Boolean mode variables and X and Y are arithmetic variables.

The operator precedence list is extended in Table 5.4 to include relational operators and Boolean operators. Note that all arithmetic operators are higher in precedence than the relational operators and the Boolean operators. Thus, arithmetic expressions are evaluated before the relations involving those expressions. After

Table 5.4 Precedence Table

| | | | | |
|---|---|---|---|---|
| .ABS. | + (unary) | .N. | .LS. | .RS. |
| .A. | | | | |
| .V. | .EV. | | | |
| .P. | | | | |
| - (unary) | | | | |
| * | / | | | |
| + (binary) | - (binary) | | | |
| .G. | .GE. | .L. | .LE. | .E. .NE. |
| .NØT. | | | | |
| .AND. | | | | |
| .ØR. | .EXØR. | | | |
| .THEN. | | | | |
| .EQV. | | | | |

examining the precedence list, it can be seen that none of the parentheses were needed in the examples above. Further examples of Boolean expressions and their evaluation are given in Table 5.5.

It should be noted that such constructions as

```
0 .LE. I .LE. 10
0 .LE. I .AND. .LE. 10
```

are not legal Boolean expressions. A correct expression, the value of which is true when i is in the range

$$0 \leq i \leq 10$$

is

```
0 .LE. I .AND. I .LE. 10
```

As with algebraic expressions, two or more Boolean expressions may be equivalent in the sense that they have the same value for all possible values of the operands involved. In fact, given any expression, there are an infinite number of other

Table 5.5 Examples of Boolean Expressions

| Expressions | Parentheses added | Value† |
|---|---|---|
| .NØT. F1 .AND. T1 | (.NØT. F1) .AND. T1 | true |
| F2 .ØR. F1 .AND. T2 | F2 .ØR. (F1 .AND. T2) | false |
| T1 .AND. T2 .ØR. F1 .AND. F2 | (T1 .AND. T2) .ØR. (F1 .AND. F2) | true |
| T1 .ØR. T2 .AND. F1 .ØR. F2 | (T1 .ØR. (T2 .AND. F1)) .ØR. F2 | true |
| .NØT. T1 .AND. .NØT. T2 | (.NØT. T1) .AND. (.NØT. T2) | false |
| T1 .AND. .NØT. T2 .ØR. T2 .AND. .NØT. T1 | (T1 .AND. (.NØT. T2)) .ØR. (T2 .AND. (.NØT. T1)) | false |

†The values shown are based on T1 and T2 being Boolean mode variables whose value is true, and F1 and F2 being Boolean mode variables whose value is false.

expressions which are equivalent to it. Some pairs of equivalent expressions are:

| | |
|---|---|
| .NØT.A .AND. .NØT.B | .NØT.(A.ØR.B) |
| A.AND..NØT.B .ØR. B.AND..NØT.A | A .EXØR. B |
| (A.ØR.B) .AND. (A.ØR.C) | A .ØR. B.AND.C |
| TØ BE .ØR. .NØT. TØ BE | 1B |

where A, B, C and TØ BE are in Boolean mode. In each case above, using the more simple expression on the right would conserve computer time. More detailed information on Boolean algebra is available from numerous sources. Mathematical works on abstract algebra usually contain a chapter on this subject and even complete books are devoted to it.[1] Considerable material on this subject is also found in most texts on computer logic, where it plays a primary role.

The Boolean substitution statement is of the same general format as the arithmetic substitution statement, namely

<bool-var> = <bool-exp>

The expression on the right of the equal sign is evaluated (as either true or false) and then the variable on the left is set equal to that value. For example,

BØØL = A .AND. B .ØR. .NØT. C

would be a Boolean substitution statement if the variables were all Boolean.

[1]F. E. Hohn, Applied Boolean Algebra, An Elementary Introduction, The Macmillan Company, New York, 1960.

Exercises

1. Construct a truth table indicating whether or not each of the following Boolean expressions is true for all possible combinations of truth values for R1, R2 and R3. (Note that there are eight possible truth value combinations of these three. If there were only two variables you would need four combinations and if there were only one variable, you would need two combinations.)

a) .NØT. R1 .ØR. .NØT. R2

b) .NØT. (R1 .AND. R2)

c) R1 .AND. .NØT. R1

d) R1 .AND. R2 .ØR. .NØT. R3

e) R1 .AND. .NØT. (R1 .ØR. R2)

f) R2 .ØR. .NØT. R2

2. The following program is supposed to calculate the quadrant (on the cartesian plane) in which a point with coordinates (A,B) falls. Explain and correct all errors.

```
READ DATA
C = 3
WHENEVER A.G.0 .AND. B.G.0, C = 1
D = A.G.0 .AND. B. .NG. 0
WHENEVER D, C = 4
D = A.LE.0 .AND. B.G.0
WHENEVER D, C = 2
INTEGER C
PRINT RESULTS C
END ØF PRØGRAM
```

3. Explain the errors, if any, in each of the following statements. Assume A, B, C, and D are Boolean mode variables, and X, Y, and Z are numeric variables.

a) A = .NØT. A

b) A = (X .ØR. Y) .G. Z

c) Z = X*Y - Y

d) A = B .NØT. C

e) D = WHENEVER X .G. Y

f) WHENEVER 1B, CØNTINUE

g) WHENEVER A .E. 1B, TRANSFER TØ START

h) WHENEVER X, Y = 1

## 5.7 The Compound Conditional

The compound conditional statement, which actually consists of a collection of statements, provides the most versatile means of controlling the order in which statements are executed in a MAD routine. Several Boolean expressions may be used in this statement to conditionally determine which of various parallel paths are to be executed.

The compound conditional statement consists of a sequence of statements which has the following general form:

```
WHENEVER <bool-exp>1
     <statement-set>1
ØR WHENEVER <bool-exp>2
     <statement-set>2
ØR WHENEVER <bool-exp>3
     <statement-set>3
.
     .
.
     .
.
     .
ØTHERWISE
     <statement-set>0
END ØF CØNDITIØNAL
```

where the following conditions must be satisfied:

1) The condition statements (i.e. the WHENEVER, ØR WHENEVER, and ØTHERWISE statements) and the terminal statement (i.e. the END ØF CØNDITIØNAL statement) delimit the statement sets of the compound conditional.

2) The first condition statement must be a WHENEVER statement and this statement may be used only to begin compound conditionals. (Note: No comma follows the Boolean expression in this statement and no other statement is contained in it. Thus, it is not the same as the conditional clause of a simple conditional statement.)

3) There may be any number (including none at all) of ØR WHENEVER condition statements followed by their statement sets. (Note: The Boolean expressions in these statements also must not be terminated by a comma.)

4) A single ØTHERWISE condition statement and its statement set may or may not occur. This condition statement, if it occurs, must be the last condition statement in the compound conditional.

5) The last statement of a compound conditional must be an END ØF CONDITIØNAL statement and this statement is used only to terminate compound conditionals.

6) The statement sets in a compound conditional statement normally consist of one or more MAD statements (each beginning on a new card and following all the rules for statements). Although these statement sets may be empty, this would seldom be useful. A few MAD statements (such as the END ØF PRØGRAM statement) are clearly not allowed in these statement sets; however, most types of MAD statements may occur. Only complete compound conditional statements and complete iteration# statements may be contained in any one of these statement sets.

The execution of a compound conditional statement is illustrated by a flowchart in Figure 5.7. Although the successive decision boxes are arranged vertically in this illustration, it is sometimes more convenient to locate them horizontally as in Figure 5.11. As shown there, the Boolean expressions of the condition statements are evaluated successively beginning with <bool-exp>$_1$ until the first of these Boolean expressions having <u>true</u> as its value is found. If this expression is <bool-exp>$_i$ then the corresponding statements making up <statement-set>$_i$ are executed. The compiler automatically inserts a transfer to the END ØF CØNDITIØNAL after each of these statement sets. The END ØF CØNDITIØNAL causes no computation, but serves as a termination for the compound conditional. Thus, the execution of the compound conditional is usually completed with the execution of only one statement set. If none of the Boolean expressions are <u>true</u>, then the statement set following the ØTHERWISE statement, if one is present, is executed, followed by execution of the END ØF CØNDITIØNAL. If none of the Boolean expressions are <u>true</u> and there is no ØTHERWISE statement, then only the condition statements and the END ØF CØNDITIØNAL are executed.

The condition statements, the END ØF CØNDITIØNAL statements and any of the statements in the statement sets may be labeled. Thus transfers to any of these statements may occur from outside the compound conditional or from other statements in the statement sets of the compound conditional. Transfers of this type, other than those within a single statement set, should generally be avoided, although their effect is well defined and they are sometimes quite valuable to the expert user. Transfers from inside a statement set of a compound conditional to statements outside the compound conditional are very commonly used. Such transfers cause a variation in the usual order of execution discussed above.

It is important to note that even if more than one Boolean expression is <u>true</u>,

only the statement set following the first Boolean expression which is true will normally be executed. Of course a transfer in this statement set, as discussed above, could cause part or all of some other statement sets to be executed. Furthermore, since <bool-exp>$_2$ is evaluated only if <bool-exp>$_1$ is false, there is an implied condition in the first ØR WHENEVER statement of

<bool-exp>$_2$ .AND. .NØT. <bool-exp>$_1$

Similarly, at each condition statement there is the implication that all of the previous expressions have been false.

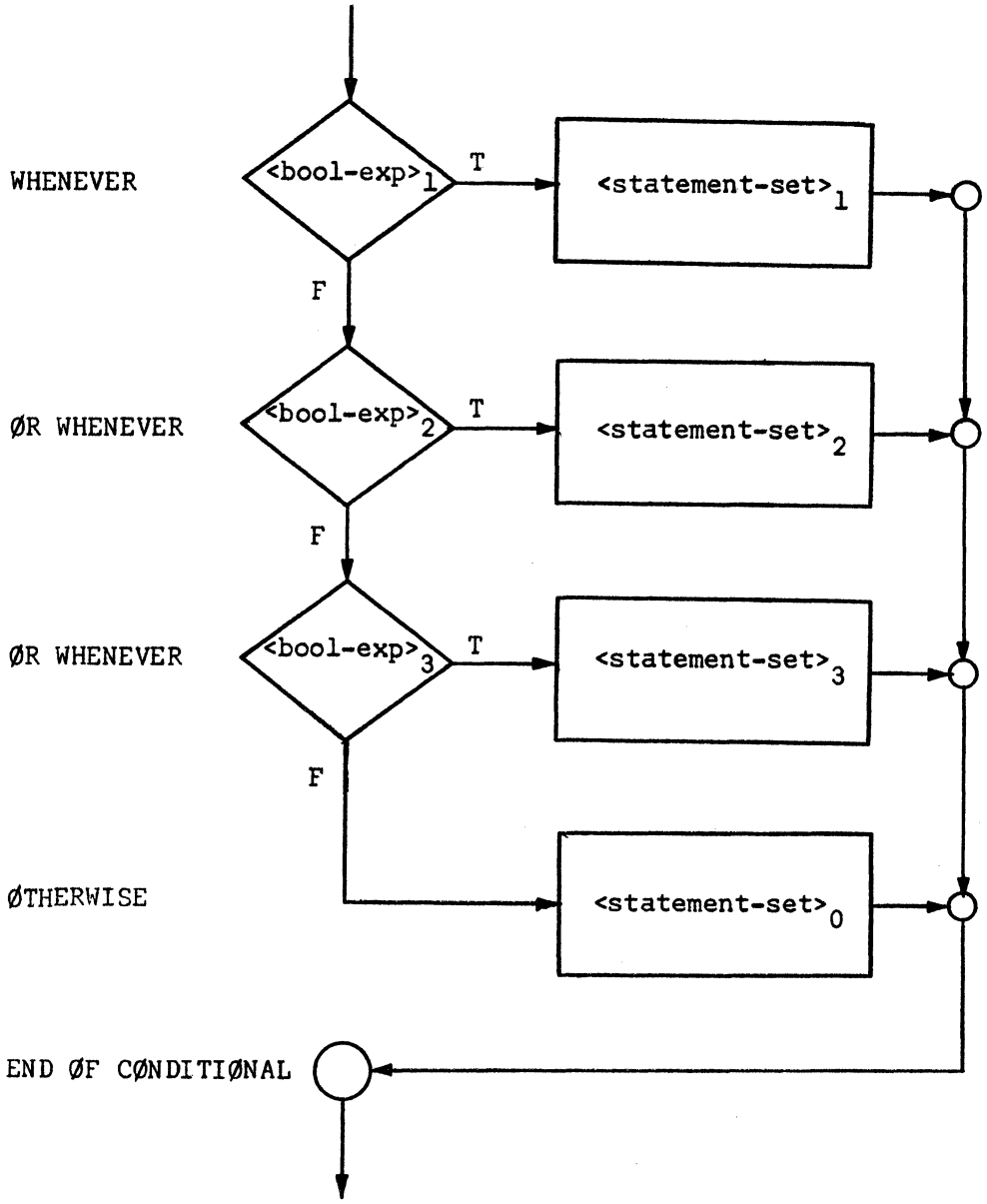


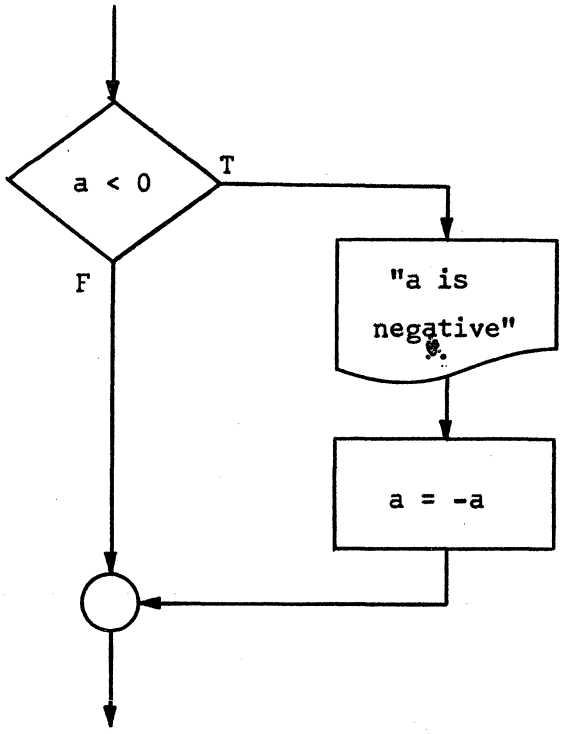

Figure 5.7 Flowchart of the Compound Conditional Statement

For ease in reading, it is convenient to indent each set of statements; we will illustrate this in the examples that follow.

The most elementary form of the compound conditional is one containing only a WHENEVER statement, its statement set, and an END ØF CØNDITIØNAL statement, such as

```
WHENEVER A .L. 0.
     PRINT CØMMENT $ A IS NEGATIVE.$
     A = -A
END ØF CØNDITIØNAL
```

The corresponding flowchart is




This form is similar to the simple conditional, except that more than one statement may be executed if the expression is true.

A second more complex example, with the corresponding flowchart, is shown in Figure 5.8.

```
WHENEVER X .LE. -1.
     Y = -1.
ØR WHENEVER X .L. 1.
     Y = X
ØTHERWISE
     Y = 1.
END ØF CØNDITIØNAL
```

Figure 5.8a

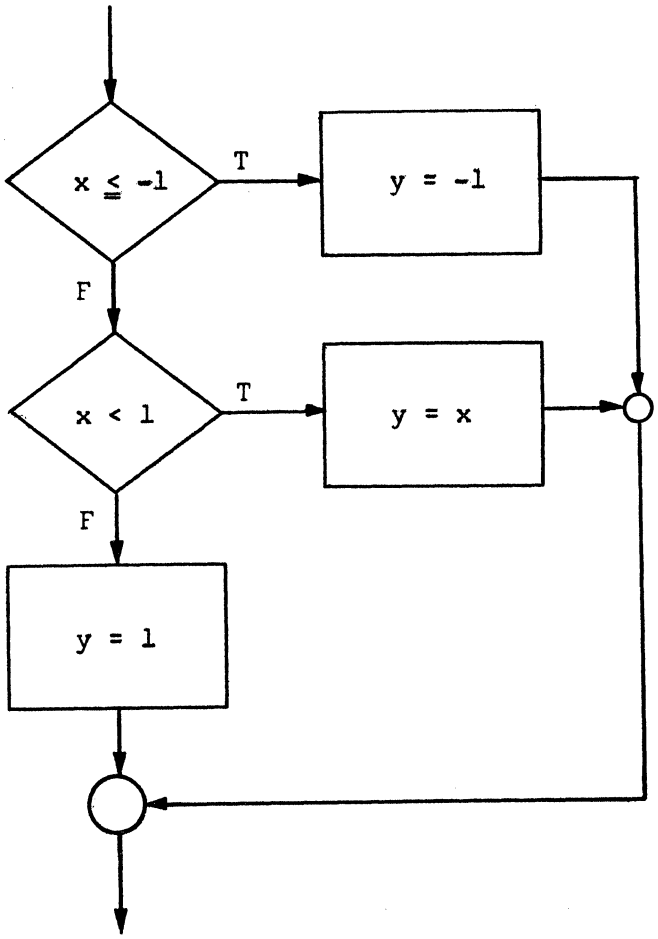



Figure 5.8b

This compound conditional evaluates the function:

$$y = \begin{cases} -1 & \text{when} \quad x \leq -1 \\ x & \text{when} \quad -1 < x < 1 \\ 1 & \text{when} \quad x \geq 1 \end{cases}$$

A third example involves the computation of income tax in an imaginary country:

> In the sovereign state of Numb, the standard unit of monetary value is the sumve. Citizens of Numb who are less than 21 years of age and those 65 or over pay no income tax. Other citizens pay a 10 percent tax on their net incomes. A taxpayer may deduct 1000 sumves from his gross income for each of his dependents including himself if he has other dependents. Taxpayers having no dependents, other than themselves, pay tax on their full incomes. Write an algorithm which will input the necessary information and print out a citizen's tax bill. Assume that the government is generous and does not require its taxpayers to pay any fractions of sumves because of the coin shortage.

Since the only information needed to compute a person's income tax in Numb is his age, number of dependents and income, it is assumed that this information is available along with the name of each taxpayer. This information is to be placed on a single card for input to the computer.

A MAD routine to make this income tax calculation and the corresponding flowchart are shown in Figure 5.9. Although this algorithm has a structure which is easily formulated in terms of a MAD compound conditional statement, the flow of control needed here varies substantially from that indicated in the standard flowchart of Figure 5.7. This variation in structure is obtained by the use of transfer statements within the statement sets of the compound conditional.

```
            NORMAL MODE IS INTEGER
            DIMENSION NAME(4)
    NEXT    READ AND PRINT DATA      NAME, AGE, DEP, INCOME
            WHENEVER AGE .L. 21 .OR. AGE .GE. 65
                PRINT COMMENT $ONO TAX, TOO YOUNG OR TOO OLD.$
                TRANSFER TO NEXT
            OR WHENEVER DEP .G. 0
                NET = INCOME - 1000*(DEP+1)
                WHENEVER NET .LE. 0, TRANSFER TO POOR
            OR WHENEVER DEP .E. 0
                NET = INCOME
            OTHERWISE
                NET = INCOME
                PRINT COMMENT $0HOW SO, NEGATIVE DEPENDENTS.$
                PRINT COMMENT $ CORRECT OR PAY FULL TAX.$
            END OF CONDITIONAL
            TAX = NET/10
            PRINT RESULTS TAX
            PRINT COMMENT $+                                        SUMVES$
            TRANSFER TO NEXT
    POOR    PRINT COMMENT $ONO TAX, TOO POOR OR TOO MANY.$
            TRANSFER TO NEXT
            END OF PROGRAM
$DATA
NAME = $DOE, DAVID D.$, AGE = 20, DEP = 0, INCOME = 2003*
NAME = $PUBLIC, JOHN Q.$, AGE = 35, DEP = 4, INCOME = 9463*
NAME = $ALGORITHM, IMA$, AGE = 29, DEP = 0, INCOME = 3460*
NAME = $SMILTZ, ROBERT C.$, AGE=31, DEP = -3, INCOME = 7432*
NAME = $TIME, FATHER$, AGE = 974, DEP = 0, INCOME = 100000000*
```

Figure 5.9a

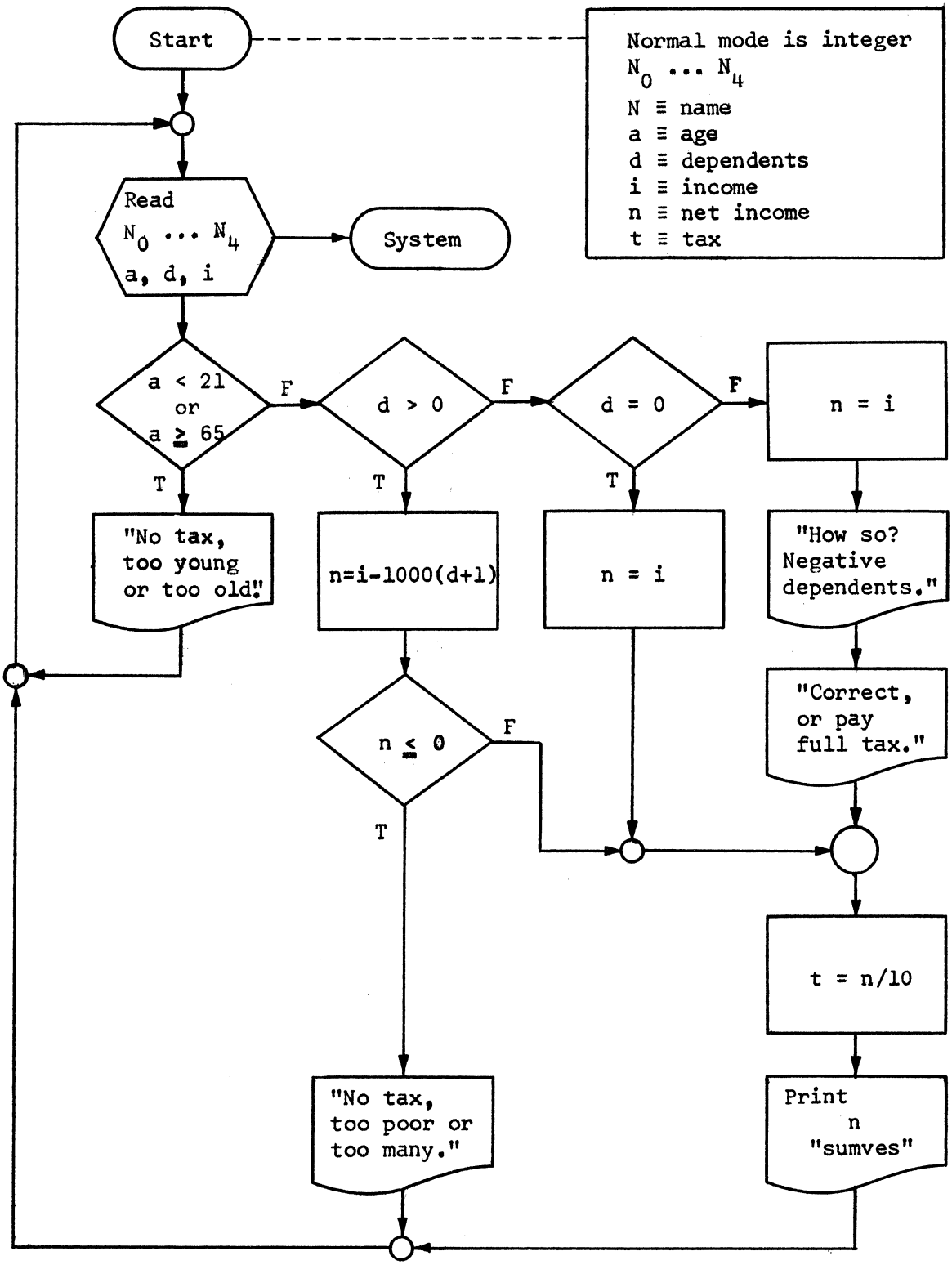

Start
Normal mode is integer
$N_0$ ... $N_4$
N ≡ name
a ≡ age
d ≡ dependents
i ≡ income
n ≡ net income
t ≡ tax
Read
$N_0$ ... $N_4$
a, d, i
System
a < 21 or a ≥ 65
F
d > 0
F
d = 0
F
n = i
T
T
T
"No tax, too young or too old."
n=i-1000(d+1)
n = i
"How so? Negative dependents."
n ≤ 0
F
"Correct, or pay full tax."
T
t = n/10
"No tax, too poor or too many."
Print n "sumves"

Figure 5.9b

The following details should be noted in the MAD routine of Figure 5.9a:

a) The vector NAME is used so that the name (up to 30 characters) of the taxpayer may be included on the data card. The name of the taxpayer appears on the output listing as a result of the READ AND PRINT DATA statement, but is not used elsewhere in the routine. The name could be printed with the results using the following statement:

```
PRINT BCD RESULTS NAME(0)...NAME(4)
```

This would require that the vector be initialized to blanks before reading each set of data; otherwise residues of previously used longer names might appear. Such complications have been omitted to simplify the routine. Problems of this type are usually done using formatting capabilities which allow the output to be placed anywhere on the page that is desired.

b) A PRINT CØMMENT statement is used with + as the carriage control character. This statement will print the word "SUMVES" on the same line as the output of the previous PRINT RESULTS statements. Thus a sample line of output might be:

```
TAX =          432 SUMVES
```

c) The transfer type simple conditional statement in the second statement set causes the flow to exit completely from the compound conditional when the value of the Boolean expression is <u>true</u>. Otherwise, it remains within the compound conditional, proceeding to the END ØF CØNDITIØNAL statement and hence out the normal exit.

d) Since the income TAX is computed from NET income using integer division by 10, any fractional part of a sumve is ignored as required by the problem.

Within any set of statements of a compound conditional, other compound conditionals may appear; this placing of one compound conditional within a portion of another is referred to as <u>nesting</u>. The only restriction on nesting is that whenever a compound conditional statement is nested inside another compound conditional statement, the entire inner compound conditional must be within a single statement set of the outer compound conditional.

As an example of nested compound conditionals, consider the routine to calculate and print the quadrant in the X-Y plane of a pair of numbers. A flowchart and the corresponding MAD routine are shown in Figure 5.10. The statement labels have been used only to allow easy reference to statements in the discussion below. They

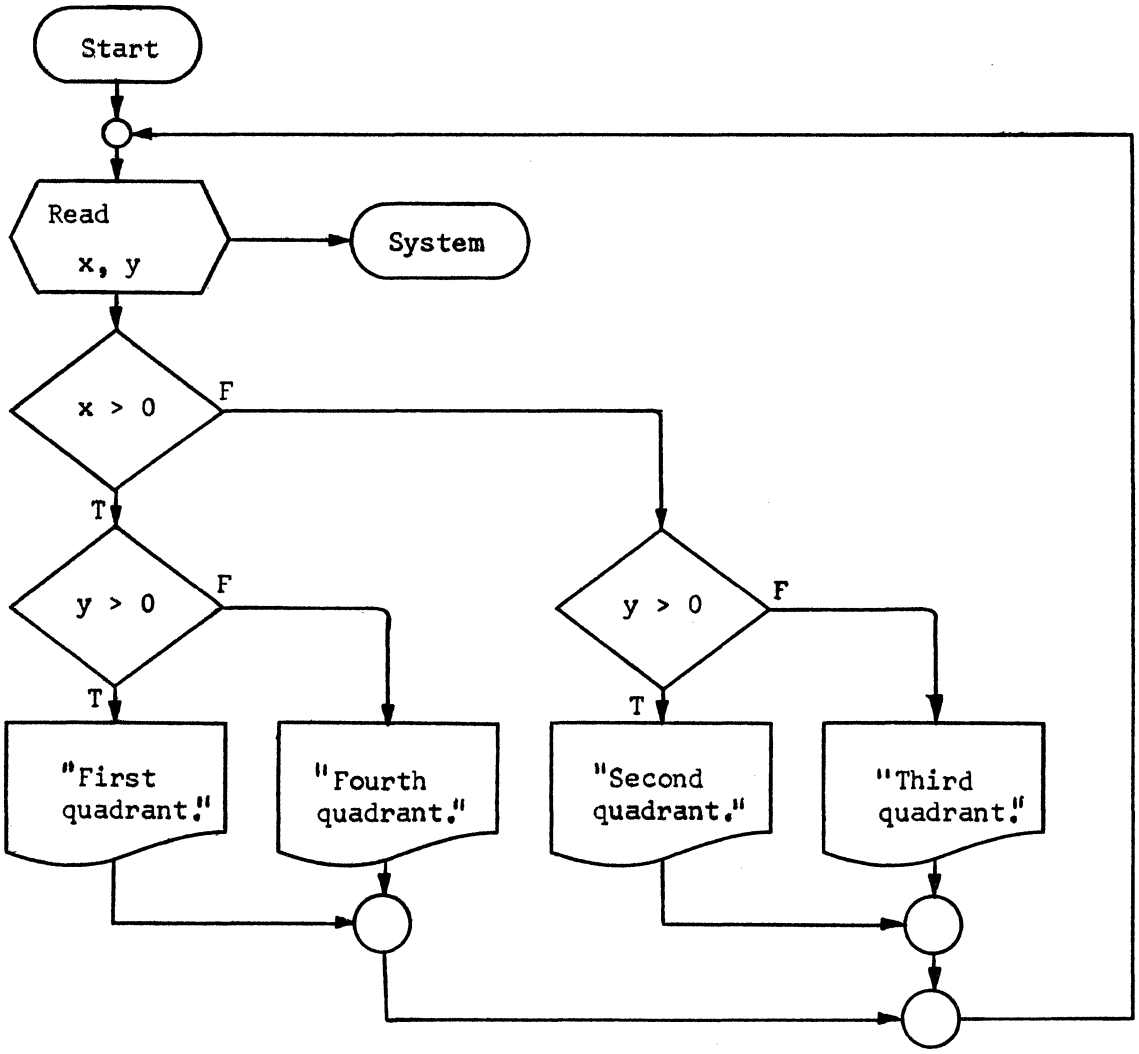



Figure 5.10a

```
          RRØUTINE TØ CALCULATE QUADRANT ØF CØØRDINATE PØINT (X,Y)
START      READ AND PRINT DATA          X, Y
A          WHENEVER X .G. 0.
B              WHENEVER Y .G. 0.
                   PRINT CØMMENT $ FIRST QUADRANT. $
               ØTHERWISE
                   PRINT CØMMENT $ FØURTH QUADRANT. $
C              END ØF CØNDITIØNAL
           ØTHERWISE
D              WHENEVER Y .G. 0
                   PRINT CØMMENT $ SECØND QUADRANT. $
               ØTHERWISE
                   PRINT CØMMENT $ THIRD QUADRANT. $
E              END ØF CØNDITIØNAL
F          END ØF CØNDITIØNAL
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 5.10b

would not be used (although they would not cause any trouble if they were used) when actually running this program on the computer.

When an END ØF CØNDITIØNAL statement is encountered by the compiler, it is matched with the last WHENEVER statement (beginning of a compound conditional) which has not been previously so used to form the limits of a compound conditional. Thus, when the compiler finds statement C, it assumes that it is the end of the conditional starting at B (not A). Next, E is associated with D and finally F is associated with A.

This routine could also be written with a single compound conditional, using a more complex Boolean expression. The flowchart for such a routine is shown in Figure 5.11a and the corresponding MAD coding is in Figure 5.11b. Note that exactly the same output is produced by both routines. Each algorithm requires the evaluation of three Boolean expressions, although this latter one involves one expression which is more complex.

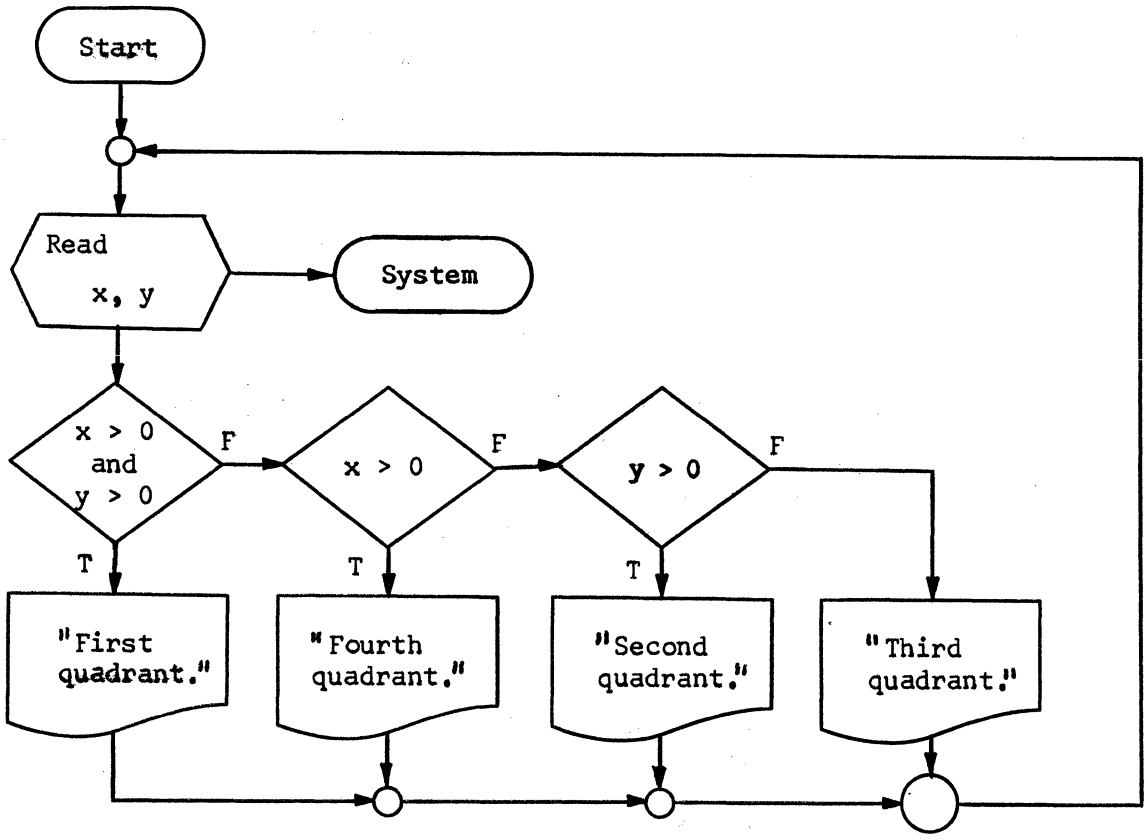



Figure 5.11a

```
          RRØUTINE TØ CALCULATE QUADRANT ØF CØØRDINATE PØINT (X,Y)
START      READ AND PRINT DATA      X, Y
           WHENEVER X .G. 0. .AND. Y .G. 0.
               PRINT CØMMENT $ FIRST QUADRANT.$
           ØR WHENEVER X .G. 0.
               PRINT CØMMENT $ FØURTH QUADRANT.$
           ØR WHENEVER Y .G. 0.
               PRINT CØMMENT $ SECØND QUADRANT.$
           ØTHERWISE
               PRINT CØMMENT $ THIRD QUADRANT.$
           END ØF CØNDITIØNAL
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 5.11b

Exercises

1. The following program, with the appropriate control cards, is run on the computer, with the input data shown. State the output that will be produced. (Include only that part of the output caused by the READ AND PRINT DATA statement and the several PRINT statements, not such things as identification, listings, maps, or accounting information).

```
START      READ AND PRINT DATA          A, B, C, X
           I = 0
           WHENEVER A.P.2 - B*C .LE. X*X/2.
               I = 3.0 + B - C.P.1./2.
               PRINT RESULTS I, B
           ØR WHENEVER A .E. 0
               I = I + 1
               PRINT RESULTS I
               PRINT CØMMENT $ I = I + 1 $
           ØR WHENEVER .ABS.X.P.2 .G. 0
               PRINT RESULTS Y
           ØTHERWISE
               TRANSFER TØ START
           END ØF CØNDITIØNAL
           PRINT CØMMENT $ END ØF PRØGRAM $
           TRANSFER TØ START
           END ØF PRØGRAM
$ DATA
A=1., B=1., C=1., X=0. *
A=0., B=-1., C=1., X=0.*
A=1., B=-1., C=2., X=0.*
A=0., B=0., C=6.0, X=1.*
A=-1., B=0., C=6.0, X=1.*
A=2., B=2., C=-3., X=1.*
```

2. Rewrite the MAD program shown in Figure 5.9 using nested compound conditional statements. Incorporate all actions taken in the different circumstances within the statement sets of the main compound conditional statement.

3. Recall that in a compound conditional only the first true relation causes its immediately following set of statements to be executed. Hence, all relations after the first one have an implied additional condition. For example, consider:

```
WHENEVER A .G. B+1
    <statement-set>1
ØR WHENEVER A .G. B
    <statement-set>2
ØTHERWISE
    <statement-set>0
END ØF CØNDITIØNAL
```

Suppose A .E. B+1 is true, then what statement set is executed?

The actual condition for statement set 2 to be executed is that the relation A .G. B .AND. .NØT.A.G.B+1 be true.

Now for the problem proper.

a) Given the following compound conditional, display four sets of numbers, A,B,C that will cause the four output results.

b) State the actual complete condition that must be satisfied in each of the four cases.

The compound conditional is:

```
WHENEVER A.E.0. .AND. B.E.0. .AND. C.E.0.
    PRINT CØMMENT $ X ARBITRARY $
ØR WHENEVER A.E.0. .AND. B.E.0.
    PRINT CØMMENT $ NØ SØLUTIØN $
ØR WHENEVER A .E. 0.
    PRINT CØMMENT $ IT'S ØNLY LINEAR $
ØTHERWISE
    PRINT CØMMENT $ GET TØ WØRK $
END ØF CØNDITIØNAL
```

## 5.8 A Sample Routine for the Solution of Quadratic Equations

We will now consider our first major example, namely an algorithm for the calculation of the roots of quadratic equations. If the equation is expressed in the form

$$a_2x^2 + a_1x + a_0 = 0$$

then, by the quadratic formula, the roots are:

$$x_1 = \frac{-a_1 + \sqrt{a_1^2 - 4a_0a_2}}{2a_2} \qquad x_2 = \frac{-a_1 - \sqrt{a_1^2 - 4a_0a_2}}{2a_2}$$

Although it might appear that one need only write the equivalent MAD expressions and thus obtain an almost trivial algorithm, there are three complications that make this straightforward procedure undesirable. First, complex roots are possible and they cannot be handled directly in MAD. In that case, the real and imaginary parts of the roots are normally calculated and so each root will involve two numbers. Second, if one of two real roots is nearly zero and the other root is several orders of magnitude larger, round off may be a serious problem in calculating the smaller root. As an example, consider the problem discussed in Chapter Two, namely that of finding the roots of

$$x^2 + 10^4x + 1 = 0$$

$$x_1 = \frac{-10^4 + \sqrt{10^8 - 4}}{2}$$

However, $10^8 - 4$ would be rounded to $10^8$ in most machines and the result would thus be $x_1 = 0$. The actual value of the root is very close to $-10^{-4}$. This problem can be alleviated, as is suggested in Chapter Two, by first finding $x_2$, the root that is larger in magnitude and making use of the property that $x_1x_2 = a_0/a_2$. Thus, the algorithm should, in the case of real roots, calculate the larger root using the formula and then derive the other root from it. Finally, the algorithm should be designed to give a correct response in the degenerate cases. For example, when $a_2 = 0$ the quadratic formula cannot be used, since it would require division by zero.

The flowchart and corresponding MAD program for an algorithm which takes these complications into account is shown in Figure 5.12.

```
$COMPILE MAD, EXECUTE
          RPROGRAM FOR THE SOLUTION OF A(2)*X*X + A(1)*X + A(0) = 0
           DIMENSION A(2)
START      READ AND PRINT DATA     A(0)...A(2)
N1         WHENEVER A(2) .NE. 0.
N5             D = A(1)*A(1) - 4.*A(0)*A(2)
N3             WHENEVER D .E. 0.
N6                 PRINT COMMENT $ DOUBLE REAL ROOT, X $
N7                 X = -A(1)/(2.*A(2))
N8                 PRINT RESULTS X
N9             OR WHENEVER  D .G. 0.
                   PRINT COMMENT $ REAL ROOTS, X1 AND X2 $
N13                WHENEVER A(1) .L. 0.
                       X1 = (-A(1) + D.P..5)/(2.*A(2))
                   OTHERWISE
                       X1 = (-A(1) - D.P..5)/(2.*A(2))
N14                END OF CONDITIONAL
                   X2 = A(0)/(A(2)*X1)
                   PRINT RESULTS X1, X2
N11            OR WHENEVER A(1) .E. 0.
                   PRINT COMMENT $ IMAGINARY ROOTS, I*Y1, I*Y2 $
                   Y1 = (-D).P..5 / (2.*A(2))
                   Y2 = -Y1
                   PRINT RESULTS Y1, Y2
N12            OTHERWISE
                   PRINT COMMENT $ COMPLEX ROOTS, X1+I*Y1, X2+I*Y2 $
                   X1 = -A(1)/(2.*A(2))
                   X2 = X1
                   Y1 = (-D).P..5/(2.*A(2))
                   Y2 = -Y1
                   PRINT RESULTS X1, Y1, X2, Y2
N4             END OF CONDITIONAL
           OR WHENEVER A(1) .NE. 0.
               PRINT COMMENT $ LINEAR EQUATION, ONE ROOT, X $
               X = -A(0)/A(1)
               PRINT RESULTS X
           OR WHENEVER A(0) .NE. 0.
               PRINT COMMENT $ NO VALUE OF X SATISFIES THIS EQUATION. $
           OTHERWISE
               PRINT COMMENT $ ANY VALUE OF X SATISFIES THIS EQUATION. $
N2         END OF CONDITIONAL
N10        TRANSFER TO START
           END OF PROGRAM
$DATA
A=0.,0.,0.*
A=1.,0.,0.*
A=0.,1.,0.*
A=1.,1.,1.*
A=1.,0.,1.*
A=0.,1.,1.*
A=1.,2.,1.*
A=2.,-3.,1.*
A=-3.,5.,1.*
```

Figure 5.12a

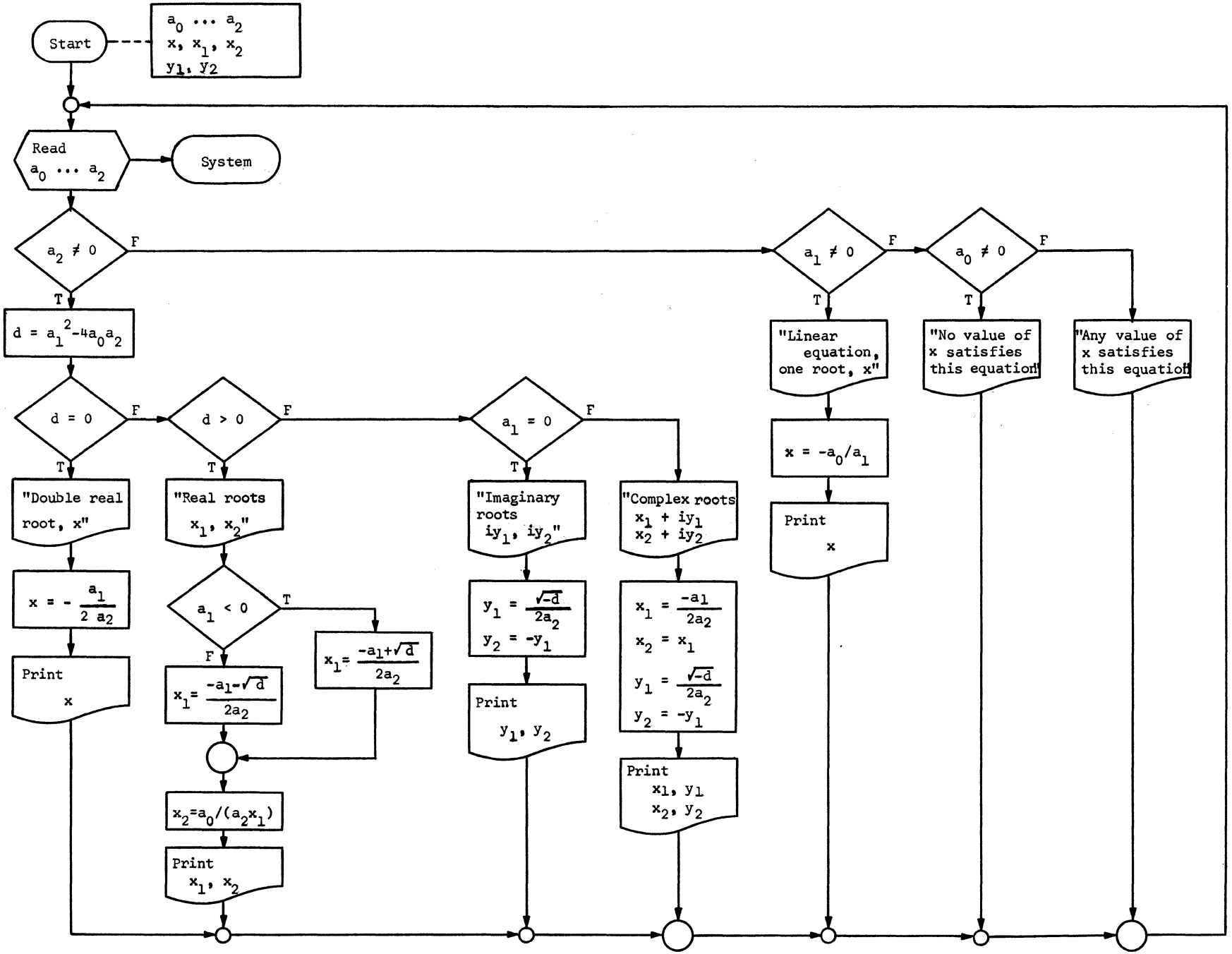

Start
$a_0 \cdots a_2$
$x, x_1, x_2$
$y_1, y_2$
Read
$a_0 \cdots a_2$
System
$a_2 \neq 0$
F
T
$d = a_1^2 - 4a_0a_2$
$a_1 \neq 0$
$a_0 \neq 0$
"Linear equation, one root, x"
"No value of x satisfies this equation"
"Any value of x satisfies this equation"
$x = -a_0/a_1$
Print x
d = 0
d > 0
$a_1 = 0$
"Double real root, x"
"Real roots $x_1, x_2$"
"Imaginary roots $iy_1, iy_2$"
"Complex roots $x_1 + iy_1$ $x_2 + iy_2$"
$x = -\frac{a_1}{2a_2}$
$a_1 < 0$
$x_1 = \frac{-a_1 + \sqrt{d}}{2a_2}$
$x_1 = \frac{-a_1 - \sqrt{d}}{2a_2}$
$y_1 = \frac{\sqrt{-d}}{2a_2}$
$y_2 = -y_1$
$x_1 = \frac{-a_1}{2a_2}$
$x_2 = x_1$
$y_1 = \frac{\sqrt{-d}}{2a_2}$
$y_2 = -y_1$
Print $y_1, y_2$
Print $x_1, y_1$ $x_2, y_2$
$x_2 = a_0/(a_2x_1)$
Print $x_1, x_2$

Figure 5.12b

In the algorithm presented here, we have treated separately the following cases (where $d = a_1^2 - 4a_0a_2$):

1) Two distinct real roots ($a_2 \neq 0$, $d > 0$).
2) A double real root ($a_2 \neq 0$, $d = 0$).
3) Conjugate imaginary roots ($a_2 \neq 0$, $a_1 = 0$, $d < 0$).
4) Complex conjugate roots ($a_2 \neq 0$, $a_1 \neq 0$, $d < 0$).
5) Linear equation, one root ($a_2 = 0$, $a_1 \neq 0$).
6) No value of x satisfies the equation ($a_2 = a_1 = 0$, $a_0 \neq 0$).
7) Any value of x satisfies equation ($a_2 = a_1 = a_0 = 0$).

The subscripted variables $x_1$, $x_2$, $y_1$ and $y_2$ were used in the flowchart. However, since only a small number of elements of each vector are required in the routine, and since they are never read or printed as a block, it is convenient and efficient to replace these variables with X1, X2, Y1 and Y2 in the MAD routine.

All instructions labeled N1, N2, etc. are labeled so as to be identifiable in this discussion. Extra statement labels do not interfere with the operation of the program (although you should in general, use only those you need).

All of the statements from N1 through N2 (nearly the whole program) comprise one compound conditional statement. A second compound conditional, statements N3 through N4, is nested within the first, and a third, N13 through N14, is nested within the second (as well as the first).

As an example of the operation of this routine, if A(2) = 1., A(1) = 2., and A(0) = 1., and thus D = 0., the following occurs: The Boolean expression in N1, A(2).NE.0., is evaluated. Since its value is <u>true</u>, N5 is executed and D is calculated to be 0. The Boolean expression in N3 is then evaluated and its value is found to be <u>true</u>. Statements N6, N7, and N8 are thus executed, resulting in the printout

```
DØUBLE REAL RØØT, X
          X = -1.000000
```

Since all the instructions in the statement set following the WHENEVER, N3, have been executed, control is transferred to the END ØF CØNDITIØNAL of that compound conditional statement, N4. However, since N4 is the last statement of a statement set in the outer compound conditional (N1 to N2), control is transferred to the END ØF CØNDITIØNAL, N2. Thus, all instructions from N9 to N2 are skipped in this case.

Although D does not explicitly appear in the Boolean expression of statement N11, it is implicitly involved because of its occurrence in the earlier condition statements of that compound conditional. Thus, the effective expression in

statement N11 is

.NØT.D.E.0. .AND. .NØT.D.G.0. .AND. A(1).E.0.

more simply

D.L.0. .AND. A(1).E.0.

Notice that the same variables are not necessarily involved in all of the condition statements of a compound conditional. For example, it was not necessary to use A(1) in the decisions of statements N3 and N9 merely because it was needed in statement N11.

## Exercises

1. If the following program and the given sets of data were run on the computer, what printed output would result? Include only that part of the printout resulting from PRINT statements in the program.

```
START       READ AND PRINT DATA          A, B, C
            WHENEVER B.P.2-4.*A*C .GE. 0.
                RT = (B.P.2-4.*A*C).P..5
                PRINT CØMMENT $ RT IS REAL $
                PRINT RESULTS RT
            ØTHERWISE
                IRT = (4.*A*C-B.P.2).P..5
                PRINT CØMMENT $ RT IS IMAGINARY, RT = I*IRT $
                PRINT RESULTS IRT
            END ØF CØNDITIØNAL
            TRANSFER TØ START
            END ØF PRØGRAM
$ DATA
A = 1., B = 2., C = 1. *
A = 1., B = 2., C = 2. *
A = 1., B = 2., C = .5 *
```

2. A compound conditional can be rewritten using a transfer type simple conditional for each WHENEVER and ØR WHENEVER statement, and an unconditional transfer at the end of each statement set. For example, the program in problem 1 could have been written (omitting comments):

```
START       READ AND PRINT DATA
            WHENEVER B.P.2-4.*A*C .L. 0., TRANSFER TØ T1
                RT = (B.P.2-4.*A*C).P..5
                PRINT RESULTS RT
                TRANSFER TØ T2
T1              IRT = (4.*A*C-B.P.2).P..5
                PRINT RESULTS IRT
T2          CØNTINUE
            TRANSFER TØ START
            END ØF PRØGRAM
```

Using simple conditionals in this form rewrite the sample program developed in this section for the solution of the quadratic equation.

3. The following MAD main routine contains many errors and bad usages. It is supposed to find the sum of N floating point mode numbers A(0)...A(N-1) where the A's and N are read as data. You should list and explain how to correct the following:

1) All syntactic errors, i.e. those errors that will prevent the routine from compiling.

2) All errors in data.

3) All logical errors, i.e. those errors that will prevent the routine from getting the correct answer.

4) All bad usages, i.e. those things that do not fit in categories 1, 2, or 3, but do cause the machine more work or lead to confusion on the part of the reader.

Assume that the appropriate control cards precede the program.

```
            INTEGER I, N
            DIMENSIØN A(9)
END         READ AND PRINT DATAS
            S = 0.
            I = S
A           WHENEVER I.LE .N, TRANSFER TØ B
            SUM = SUM + A
            I = I + 1
            TRANSFER TØ A
B           PRINT RESULTS SUM
            TRANFSER TØ END
            END ØF PRØGRAM
$DATA
A=1,2,3,4,5,6,7,8,9,10,N=10*
A=0.,1.,2.,-1.,3.6,4.,9.,6.,5.,2.,4.*
```

Chapter Six

# REPETITIVE PROCESSES

One of the most important characteristics of computers is their ability to retain and perform the same set of instructions repetitively. If the user were forced to give an instruction to the computer for each small operation, or even for each set of operations (as he does in a compiler language), he would be able to do his work only slightly more efficiently than he could do it by hand or on a desk calculator. Furthermore, a typical modern computer can execute more than a hundred thousand instructions per second, whereas its memory can hold only a few tens of thousands of instructions. Thus, it can usually execute as many instructions as can be stored simultaneously in its entire high speed memory in only a fraction of a second. Since these machines usually cannot load their instructions into memory as quickly as they can execute them, most of the time would be wasted simply waiting for instructions, were it not for repetition. Thus, in order for a computer to operate efficiently, some of the instructions must be executed many times once they have been loaded into memory. On the other hand, the accomplishing of repetition is not an end in itself since, as we have seen in Chapter Five, infinite loops may occur. Useful repetitions normally involve repeating the same set of instructions on different operands, although even this condition does not insure the production of meaningful results. The goal then is to learn how to set up the constructive reuse of instructions in such a way that our problems can be solved efficiently.

In this chapter, we will first describe the different types of repetition that are commonly used. The primary emphasis of the chapter is on loops, the successive repetition of a set of statements within a routine. The structure of such loops, the conditions which may be used to terminate them, and some examples of algorithms in which loops are used are then discussed. We next introduce the iteration statements of MAD. These compound statements provide an efficient way to implement loops. Two major areas of numerical mathematics are also introduced in this chapter. The first area is concerned with finding the roots of a function and utilizes a class of loops known as iterations. The second area is numerical integration and differentiation. The nesting of loops is also discussed and illustrated with examples.

## 6.1 Repetition in Routines

There are three types of repetition that normally take place in using a computer. They are:

1) The use of entire programs or routines on more than one occasion.
2) The performance of the same calculations on more than one set of input data as the result of a single request.
3) The repeated execution of a set of statements within a routine.

These types have many common characteristics, as we will see in this section.

Useful algorithms are often written so that they will solve a large set of similar problems. In order to make such algorithms available to others, complete and accurate documentation should be prepared, including verbal instructions on how to use them, a description of the methods employed, program listings, flowcharts and other useful material. Such algorithms are most commonly implemented in the form of subroutines, although they may be designed as complete programs. The most frequently used of the subroutines are made available in the system library and may be used by simply writing an appropriate statement. For example, it is usually not necessary to write a routine for the calculation of trigonometric functions, logarithms, or exponentials when such a mathematical function is needed. The basic input-output functions, as well as the compilers, are also handled in a similar way. Other less common algorithms are available on punched chards or magnetic tape with suitable descriptions as to their use usually being available in manuals or on microfilm. Thus, a large number of programs and routines are available to be used repeatedly without each individual user having to develop them or even understand the details of their operation. Large complex algorithms are often organized into smaller algorithms some of which may be implemented as subroutines. This is especially desirable when these smaller algorithms are needed in several places in the larger one. The writing of such subroutines in MAD is the subject of Chapter Seven.

The second type of repetitive process --the execution of a program for many sets of data-- has been illustrated in most of the examples that have been presented in earlier chapters. Routines of this type are exemplified by the flowchart of Figure 6.1. A data set is read, the program is executed for that data, the output is produced, and control is then returned to the beginning of the routine for the reading and processing of a new data set. The program is terminated when all data sets have been processed. In algorithms having alternate paths, there may be several input and output steps with alternate transfers back to various input steps

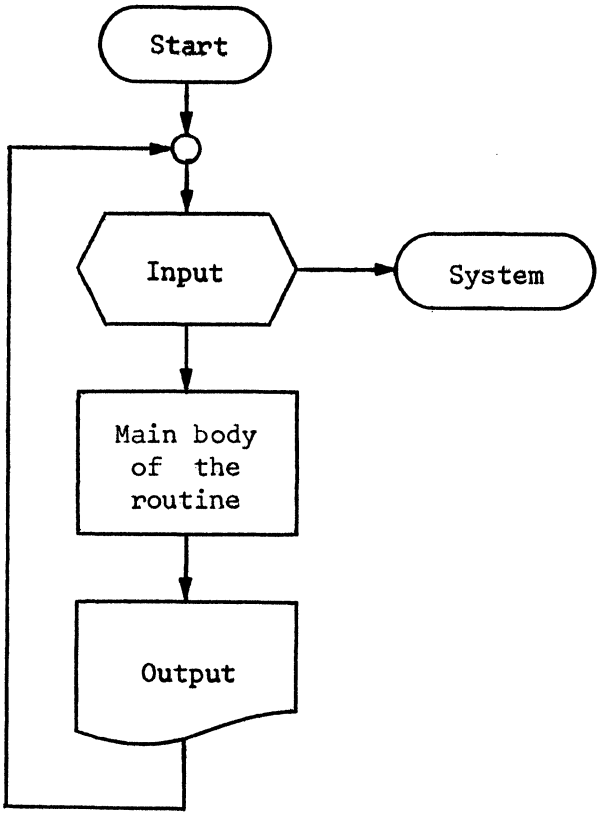



Figure 6.1

(e.g. Figure 5.3). Routines of this type will not be discussed further, since they require only the use of unconditional transfer statements to create the repetition and since the input statements automatically take care of the termination of the routine when there are no further data sets to be processed.

The third type of repetition, called the loop, is the main topic of this chapter. In this form of repetition, a set of steps within an algorithm is repeated successively during the calculations until some explicitly stated condition is met. The set of statements repeated, including those testing whether the terminating condition has been satisfied, constitute the loop. Three properties are associated with all loops. First, there must be some means of delimiting the steps that are to be repeated, that is, there must be some indication of which steps are to be included in the loop. The algorithm containing a loop that was discussed in Section 1.6, namely the algorithm for the sum of n numbers, had a transfer as the last step in the loop. This transfer returned control to the beginning of the set of steps that was to be repeated. Thus, the steps in the loop were delimited by a transfer

as the last statement and a labeled step as the first statement. All loops are basically delimited in this way, although, as is the case in the iteration statement in MAD (Section 6.3), the transfer and the labeling need not be explicit. Second, the initialization of some of the variables used must usually be accomplished before entering the loop. The algorithm in Section 1.6 required the initialization of i and s. They were then modified during the execution of the loop. The third property that all useful loops must possess is some provision for their termination. A condition or test is usually established, the satisfaction of which results in the termination of the loop. Unless the test is satisfied before entering the loop, the variables involved in the test must change during the execution of the loop, so that the terminating condition is eventually satisfied. Although these variables need not change during each pass of the loop, the loop would never be terminated unless a change occurred in some variable each time through the loop. The "infinite loops" discussed in Chapter Five either have no provision for termination or the condition provided is never satisfied. In the algorithm for the sum of n numbers, the terminating condition was that there were no more numbers to be added, i.e. that i was greater than or equal to n. In Section 6.2, we will discuss the test in more detail.

In addition to these properties, many loops also use an index. An index is a variable that is linearly dependent upon the number of times that the steps in a loop have been executed. It is often equal to the number of times the computations in the loop have been performed, but, more generally, it need only satisfy the linear relation

$$v = i + n \cdot s$$

where v is the index, n is the number of times that the computations of the loop have been executed, and i and s are any constants (positive, negative or zero; integer or floating point). A loop need not use an index at all, or it may have one or more indices. An index, v, is initialized to the value i before entering the loop and it is stepped or incremented by s during each execution of the loop, i.e. s is added to its value, thus increasing or decreasing v. The incrementing normally takes place at the end of the loop, although it may occur anywhere. The index in the algorithm of Figure 1.8, is just equal to the number of times that the computations in the loop have been executed and thus i = 0 and s = 1 in the equation above.

Notice, finally, that the second type of repetition, namely the use of an entire algorithm on many sets of data has the same structure as a loop, with the terminating condition being implicit, namely, the exhaustion of the data. Because of its simple structure, we will not include that case when we discuss loops.

## 6.2 Structure and Termination of Loops

Several basic structures are commonly used for loops. The loop described in Section 1.6 is typical of one of these structures; a general flowchart for this structure is shown in Figure 6.2. Before entering the loop, the indices and other variables that must be preset are initialized. In this particular structure, the test for termination is the first step executed in the loop itself. The test is followed by the calculations, which include the incrementing of any indices and modification of other variables. Control is then transferred back to the test at the beginning of the loop. The iteration statement in MAD creates a loop with this structure.

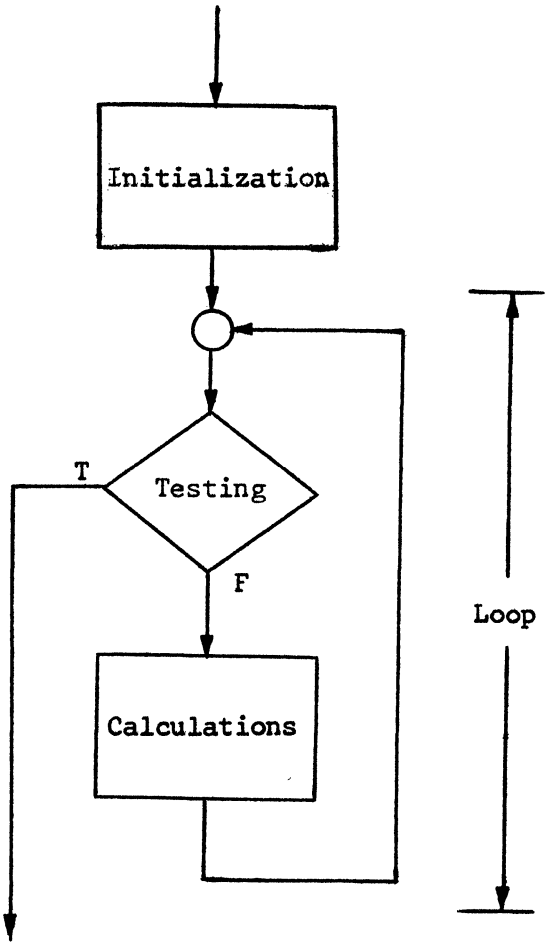



Figure 6.2

A second structure, in which the test is placed at the end of the loop, is illustrated in Figure 6.3. The most significant difference between this and the

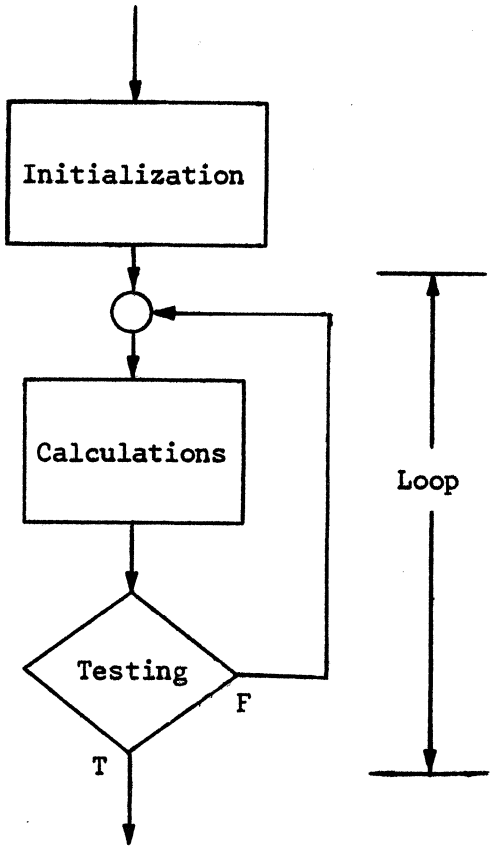



Figure 6.3

previous structure is that the calculations in the loop are always performed at least once in this case, even if the terminating condition is satisfied before entering the loop. This arrangement has the disadvantage that it often makes it necessary to test for and treat this case separately. This structure is found in the iteration statement of the FORTRAN language (i.e. the DØ statement). A loop of this form can be implemented in MAD using a transfer type simple conditional by reversing the test condition so that a false Boolean expression would indicate termination of the loop.

Other structures are possible, but are less commonly used. For example, the test can be located in the middle of the calculations, rather than at the beginning or at the end. More complex structures, involving more than one test in the loop and possibly transferring to different points outside the loop, may also be used. Transfers into the middle of a loop are possible, but can cause disastrous results

unless the variables involved have been properly initialized. The need for such transfers is rare, and they should generally be avoided.

We will now discuss some of the more common classes of loops and indicate types of problems for which they are useful. The sample algorithms in this chapter will illustrate many of these types of loops.

One class of loops includes those that terminate when the value of an index which is being increased (decreased) equals or exceeds (equals or falls below) a preset limit. The limit may be a constant, a parameter that is read in as data, or a value calculated earlier in the routine. It could even be calculated in the loop; but that is less common. This limit, together with the initial value given the index and the size of the steps taken, establishes the number of times the loop is traversed. Loops of this type often involve calculations on successive elements of a vector. The algorithms discussed in Chapter One contained loops of this type. The first loop, used in the calculation of the sum of 6 numbers, had a fixed limit, whereas the algorithm for the sum of n numbers contained a loop, the terminating condition of which depended on the value of n, which was read in as data.

A second class of loops includes those for which the calculations are made for a set of values of a parameter. If the values are evenly spaced, then the parameter can be considered an index, in which case the loop would fall in the first class. An example of such a problem is the calculation of a table of the squares and cubes of all integers from 1 to 100. To solve this problem, we would usually treat i as an integer index, first setting it equal to 1 and then incrementing it by 1 each time through the loop. In this type of problem, we would likely include an output statement within the loop, printing out one line of the table each time through the loop. Rather than setting an index and incrementing it, one could provide a list of the 100 values of i for which the calculations are to be made. When the values of the parameter are not evenly spaced, this form of the loop is commonly used. For example, problems involving the calculation of tax or interest at rates of 1%, 2%, 2½%, 3%, and 5% would fall in this class.

Many problems involve finding the value of a variable for which a function of that variable satisfies some condition. For example, one might want to find the value of x such that f(x) is a maximum or is zero. Solutions to problems of this type often involve successive approximations. Such techniques are usually referred to as iterative techniques and the solutions so produced are called iterative solutions. The process of iterating toward a solution is usually terminated when a solution that is "close enough" to the actual solution has been obtained. Such techniques are often quite valuable; however, what is meant by being "close enough"

requires careful analysis in each case. Loops of this type often do not involve indices, although they may. In some cases, an alternate terminating condition is included, such that the loop is terminated if no solution is found after a fixed number of attempts. This guards against the possibility of being stuck in an infinite loop if the successive approximations diverge rather than converge to a solution, or if there is, in fact, no solution. In Section 6.4 and 6.5 we will discuss some techniques for the iterative solution for the roots of functions.

A closely related class of loops also involves the termination of a loop on the basis of a condition not associated with the index. An example of such a problem is the following: A loan of $10,000 is obtained at 6% interest to be computed monthly on the unpaid balance. The monthly payments of $100 are used first to pay the interest with the excess being applied to reduce the principal. The last payment would normally be less than $100. Calculate and print out a table indicating for each month the number of the payment, the amount of money being paid as interest, the amount paid toward reducing the principal of the loan, and the amount still owed at the end of that month. The loop is terminated when the loan is completely paid off. An index would normally be used, since it is desirable to include in each row of the table an indication of which month is involved and also to have available the time required to pay off the loan.

Another class of loops is terminated when some particular variable, usually an element of a vector or array, is found which satisfies certain conditions. If it is possible that none of the variables satisfies the condition, then an alternate termination of the loop must be provided when all elements on the list have been examined. Such loops are often used in making a search for a particular element. A slight modification of the searching problem, namely the search for all elements on the list satisfying the test, reduces the loop used to one of the first type, since it must now be traversed once for each item on the list and is thus terminated when the value of the index reaches a preset limit.

Finally, many problems involve the nesting of loops, i.e. the placing of one loop within another. We will defer the discussion of this subject until Section 6.6.

## 6.3 The Iteration Statements

The MAD iteration statements, like the compound conditional statement, are actually sets of statements. There are two forms of iteration statements, each consisting of a THRØUGH statement followed by a set of statements (possibly empty) to be executed under the control of that THRØUGH statement. A label, used to delimit

the statement set, is contained in the THRØUGH statement and also labels the last statement in the statement set. In both forms of the iteration statement, the THRØUGH statement controls the value which one variable will have on successive executions of the statement set. The two forms vary in the way this variable is controlled and in the way the loop is terminated. In the first of these, called the increment form, the variable is initialized before the statement set is first executed and incremented after each execution of the statement set. In this case, the loop is terminated when a Boolean expression, also contained in the THRØUGH statement, becomes true. In the second form of iteration statement, called the fixed list form, an explicit list of values for the variable being controlled is contained in the THRØUGH statement, and the statement set is executed once for each of these values. The increment form may be used only for arithmetic variables, whereas the fixed list form allows the control of labels, subroutines, and Boolean variables as well.

The definitions of the two forms of the iteration statement are similar and it is convenient to use some common metavariables.

Definition of initialization, scope, and next statement:

| | | |
|---|---|---|
| <initialization> | is | a set of MAD statements, often an empty set. |
| <scope> | is | a set of MAD statements, excluding only such statements as END ØF PRØGRAM. If any part of a compound statement is included, that entire compound statement must be in <scope>. |
| <next-statement> | is | any executable statement.[1] |

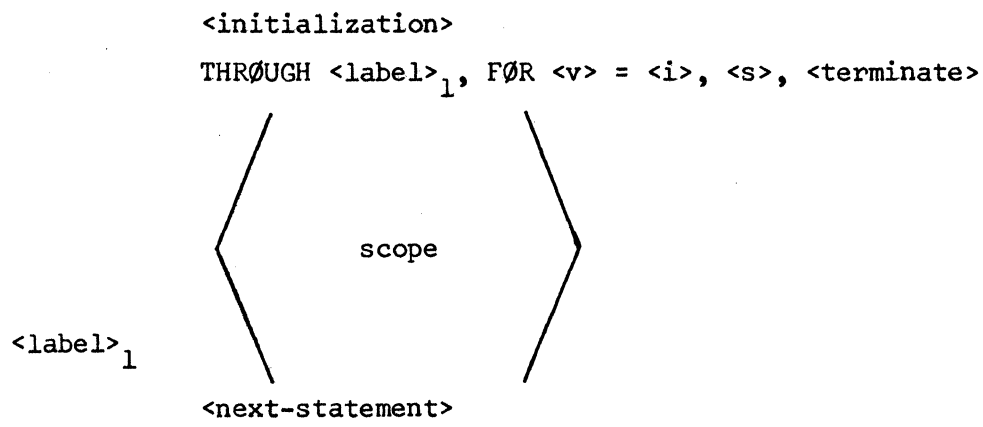
In order to define the increment form of the iteration statement, some additional metavariables are needed. The fixed list form of the iteration statement will be defined later in the section.

Definition of iteration variable, initial value, step or increment, and terminate:

| | | |
|---|---|---|
| <v> | is | <a-var> |
| <i> | is | <a-exp> |
| <s> | is | <a-exp> |
| <terminate> | is | <bool-exp> |

[1] There may be declarations and remarks between the scope and <next-statement>.

A typical structure for a loop (including the initialization of variables and the first executable statement following the loop) using the increment form of the iteration statement is as follows:




where the label indicated appears on the last statement in the scope. The last statement in the scope is often a CØNTINUE statement.

The operation of this statement can best be understood by reference to the flowcharts of Figure 6.4. In part a of the figure, we have redrawn the loop of Figure 6.2 so as to improve the topology and to conform with the terminology introduced for the iteration statement. In order to do this, we have allowed, for the first time, two entry points to a flowchart box. Both entries produce the same action, namely the evaluation of <terminate> and a branch. In Figure 6.4b, we have combined the initialization and incrementing of the iteration variable with the test of the terminating condition into a compound box, which we will refer to as an iteration box. This iteration box also has two entries, but different actions are produced for each entry. The entry at the top of the box causes the iteration variable to be initialized, whereas the entry at the bottom results in that variable being incremented. In either case, <terminate> is evaluated and the exit is determined by the value of that Boolean expression. The box labeled initialization in the flowchart includes the initialization of indices and variables other than the variable controlled in the THRØUGH statement. If it is necessary to increment indices or variables other than the variable controlled in the THRØUGH statement, this is done in the scope. Also, the circle has been included after the box for <scope> in this figure to indicate the labeled point at the end of the scope; that circle is often omitted in flowcharts.

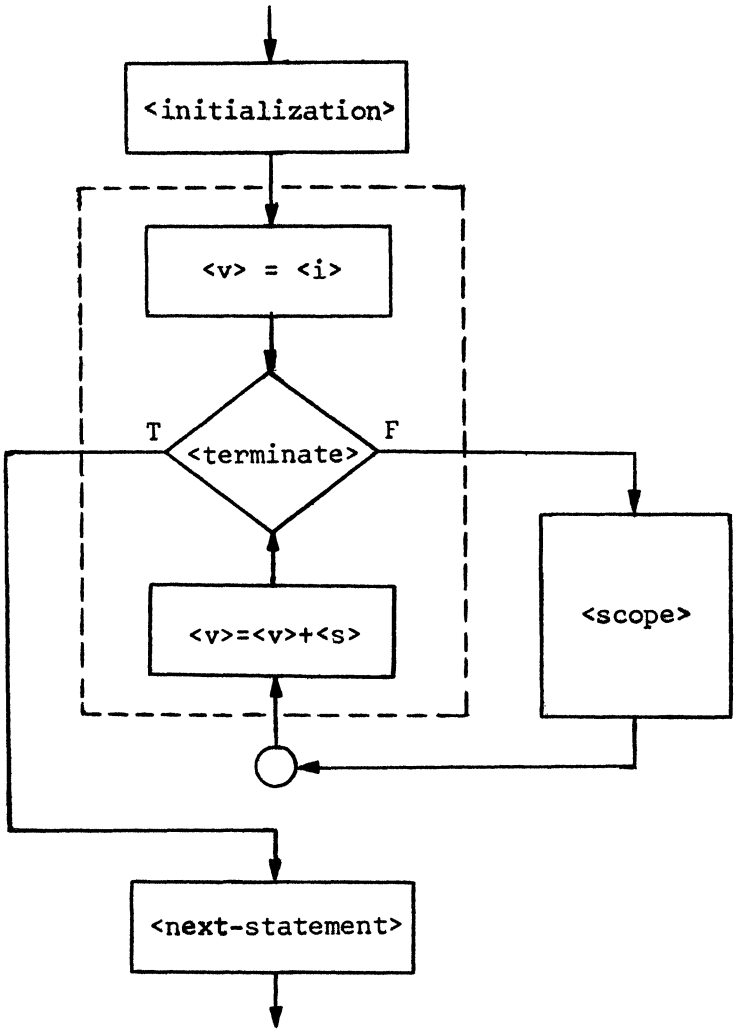



Figure 6.4a

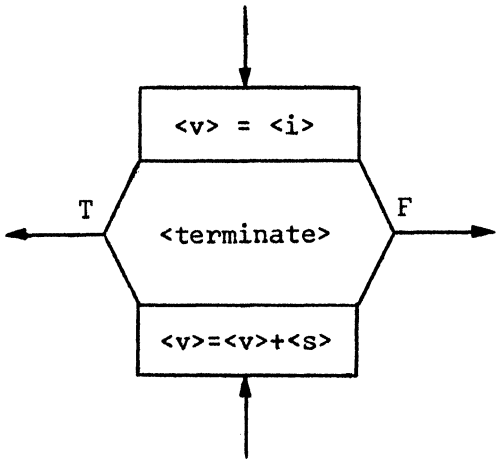



Figure 6.4b

The execution of this form of the iteration statement is as follows:

1) The iteration variable is initialized, i.e. the value of the iteration variable is set equal to the initial value (<v> = <i>).

2) The termination condition, <terminate>, is evaluated. If its value is true, the execution of the iteration statement is terminated and execution of the routine then continues with <next-statement>. Deviations from this normal order of events occur when the iteration statement is the last statement in a statement set of a compound conditional, in which case the END ØF CØNDITIØNAL is executed next, and also in the case of the common termination of nested loops (See Section 6.6). If the value of the termination condition is false, the statements in the scope are executed and step 3 is performed next.

3) The value of the iteration variable is incremented or stepped, i.e. <v> = <v> + <s> and step 2 is repeated.

The iteration variable may be incremented by a positive amount, a negative amount or zero in step 3. Furthermore, the size of the increment may be changed during the execution of the loop by modifying the variables included in the expression <s>. The iteration variable may also be modified within the scope of the iteration statement. After the normal termination of the loop, the iteration variable has the value used during the last execution of the statement labeled $\langle label \rangle_1$ incremented by <s>. The statements in the scope have not normally been executed with this value of the iteration variable.

The execution of this form of the iteration statement is exactly equivalent to that of the following loop formed using a simple conditional:

```
            <initialization>
            <v>   =   <i>
LØØP        WHENEVER <terminate>, TRANSFER TØ NEXT
           /                                    \
          /                                      \
          \                                      /
           \                scope               /
<label>1   /                                    \
            <v>   =   <v> + <s>
            TRANSFER TØ LØØP
NEXT        <next-statement>
```

It is strongly recommended that in most cases the CØNTINUE statement be used as the last statement of the scope, i.e. the labeled statement. If the last

statement in the scope is a compound conditional statement, the label may be placed on the END ØF CØNDITIØNAL statement. Extreme caution should be exercised if the CØNTINUE is omitted, particularly if nested loops are being used (see Section 6.6). The THRØUGH statement itself may be the labeled statement (i.e. the one with <label>$_1$), in which case the only action that occurs is the initialization and incrementing of the iteration variable and the testing of the termination condition.

It should be noted that whenever the THRØUGH statement is encountered during the execution of a routine, or whenever that statement is labeled and transferred to (even from within the scope of that iteration statement), the iteration variable is always initialized, i.e. execution always begins at Step 1. The variable controlled by the THRØUGH statement is incremented as a result of that statement only after the statement labeled <label>$_1$ is executed. In the equivalent loop using a simple conditional, it should be noted that the incrementing statement is not labeled, and may be reached only by executing the last statement in the scope which may of course be a CØNTINUE statement.

Although transfers into the scope of an iteration statement are legal, they involve many hazards. Since the normal initialization is not executed, all variables (including the controlled variable) have their current values, which may well be the core constant if the loop has never been executed before. Once the loop has been entered in this way, execution continues normally, i.e. the remaining statements in the loop are executed, the iteration variable is incremented and <terminate> is evaluated. Such transfers should be avoided by all but the expert user.

The scope of a loop may contain a transfer statement (conditional or unconditional) to a labeled statement outside the scope of the loop (as well as to other statements within the scope). This could, for example, be an alternate exit or termination of the loop. Once a transfer outside the scope takes place, the execution of the loop is terminated, independent of the value of <terminate>; however, execution of the loop may be resumed by providing a transfer back into the scope of the loop. This is not normally a good practice, but may sometimes be a useful technique.

The flowchart of Figure 1.8 is modified in Figure 6.5a to make use of the flowchart iteration box. The corresponding MAD routine using the iteration statement (which was originally given in Figure 1.9) is shown in Figure 6.5b. In this example, the iteration variable is an index of the loop. As in most loops, all operations on the iteration variable are included in the THRØUGH statement. The scope, in this case, contains only the statement S = S + A(I) plus a CØNTINUE statement.

Before giving further examples of the use of the iteration statement, we will define the fixed list form of the iteration statement. Two additional metavariables

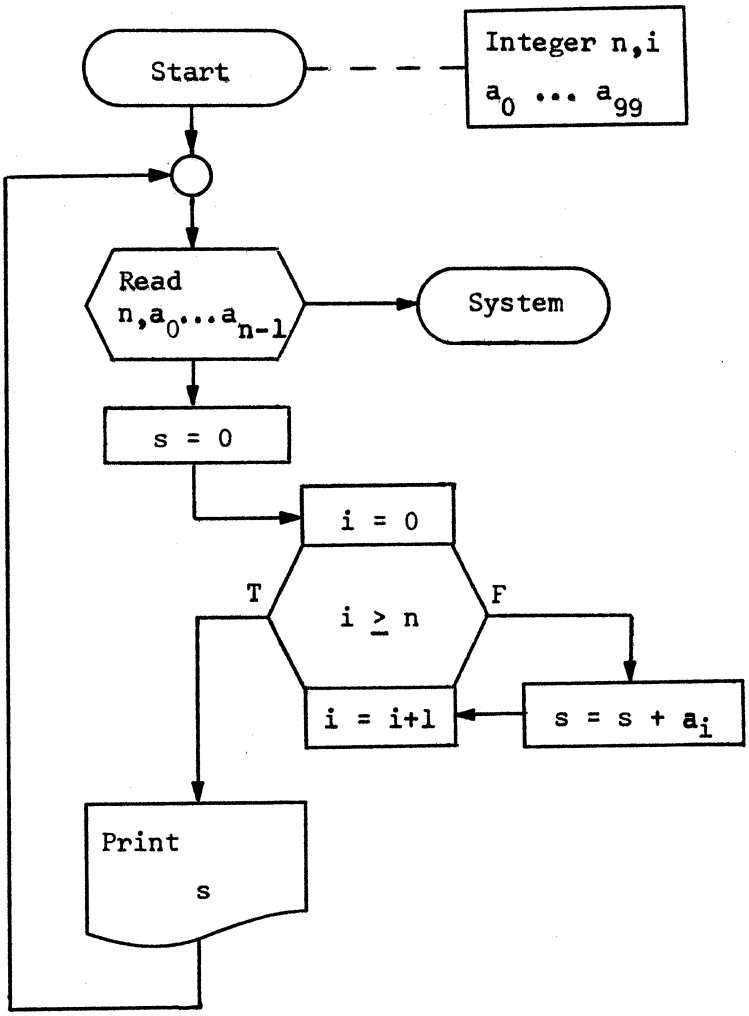



Figure 6.5a

```
          R   RØUTINE FØR THE SUM ØF N NUMBERS
           INTEGER N, I
           DIMENSIØN A(99)
START      READ AND PRINT DATA      N, A(0)...A(N-1)
           S = 0.0
           THRØUGH END, FØR I = 0, 1, I .GE. N
               S = S + A(I)
END        CØNTINUE
           PRINT RESULTS S
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 6.5b

are useful in the definition of this form of the statement.

Definition of fixed list iteration variable and value:

<it-var> is <a-var> or <bool-var> or <label-var> or <subr-var>#

<value> is <a-exp> or <bool-exp> or <label-exp> or <subr-exp>#

A typical structure for a loop using the fixed list iteration statement is as follows:

<initialization>

THRØUGH <label>$_1$, FØR VALUES ØF <it-var> = L{<value>}

<label>$_1$ scope

<next-statement>

The values on the list must correspond in mode with the iteration variable, e.g. if <it-var> is an arithmetic variable, then <value> must be an arithmetic expression. A flowchart representation of this is shown in Figure 6.6, where the labels T and F refer to the truth of the proposition that the scope has been executed once for

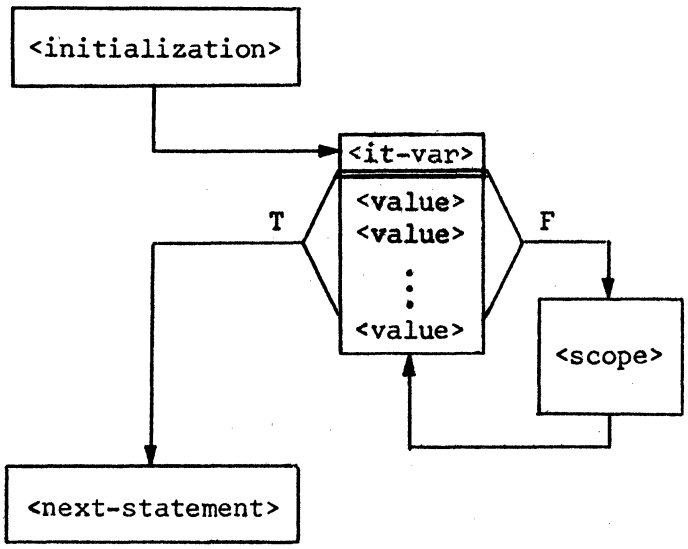



Figure 6.6

each value on the list. The operation of this form of the iteration statement may be described as follows:

> The set of statements in the scope of the iteration statement is executed successively, with the iteration variable taking on each of the values on the list in succession. After the loop is executed with the iteration variable having the last value on the list, <next-statement> is executed. (The same exceptions apply to this form of the iteration statement as to the increment form.)

As in the case of the increment form of the iteration statement, transfers into the scope of the fixed list iteration statement are allowed. If such a transfer is executed, the remaining statements in the scope are executed, using the current value of the iteration variable. If on the previous execution of the iteration statement, all values on the list were not used (i.e., if a transfer out of the scope occurred), then execution continues at that point on the list and the remaining values on the list are used. On the other hand, if this iteration statement had not been executed previously or if each value on the list had been used previously, then the loop is terminated after executing the last statement in the scope and <next-statement> is then executed. Since the normal initialization has not been executed in this case, such transfers should be avoided by all but the expert user.

The iteration variable may be an arithmetic variable, in which case the operation of this type of iteration statement is similar to that of the increment form. As in the case of that form of the statement, the variable in the THRØUGH statement may be modified during the execution of the scope. However, such modification does not affect the value that the iteration variable assumes at the beginning of the next pass through the loop.

Let us now consider an example of an algorithm and the corresponding routine involving a fixed list form of the loop, where the iteration parameter is an arithmetic variable. An algorithm to calculate and print the tax on a purchase of D dollars (read in as data in floating point mode) for tax rates of 1%, 2%, 2.5%, 3%, and 5% is shown in Figure 6.7a, with the corresponding MAD routine in Figure 6.7b. The output corresponding to the data D = 2140.50 is shown in Figure 6.7c. Although the routine is trivial, it illustrates the operation of the fixed list iteration statement and the inclusion of an output statement in the scope of a loop. The statements in the scope of the loop, the calculation of the tax and the printing, are executed first for r = .01, then for r = .02, r = .025, r = .03, and finally for r = .05. After the execution of the statements in the scope for that value, <next-statement>, i.e. TRANSFER TØ START, is executed.

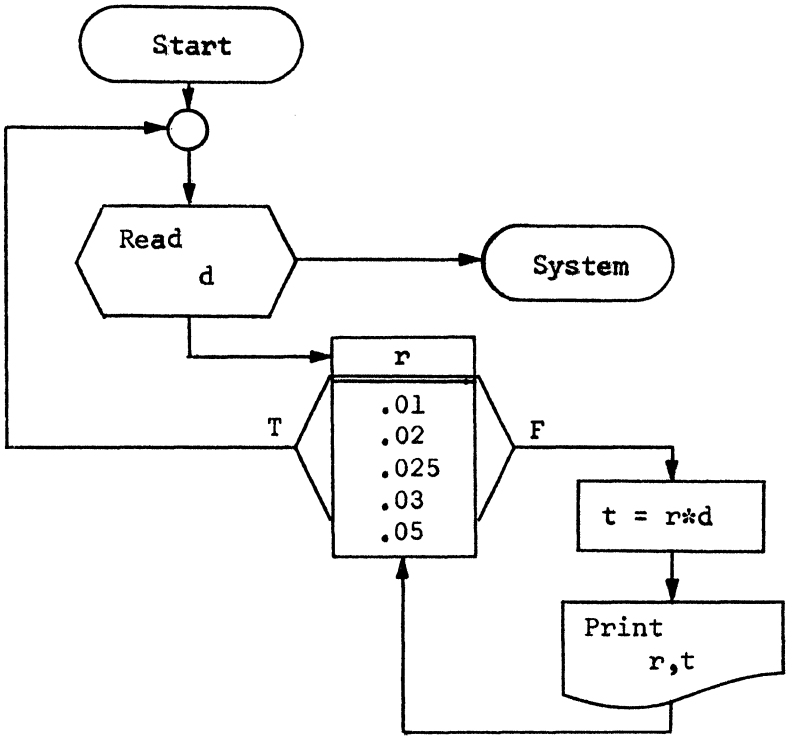



Figure 6.7a

```
          RRØUTINE TØ CALCULATE TAX ØN A PURCHASE ØF D DØLLARS
          R    AT A SET ØF RATES, R.
START      READ AND PRINT DATA          D
           THRØUGH S, FØR VALUES ØF R = .01,.02,.025,.03,.05
               TAX = R*D
               PRINT RESULTS R, TAX
S          CØNTINUE
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 6.7b

```
D=2140.50*

          R =   .010000,        TAX =   21.405000
          R =   .020000,        TAX =   42.810000
          R =   .025000,        TAX =   53.512500
          R =   .030000,        TAX =   64.215000
          R =   .050000,        TAX =  107.025000
```

Figure 6.7c

In the fixed list form of the iteration statement, the iteration variable may be in statement label mode, in which case each element of <value> must also be a label. An example of such an iteration statement would be

```
      STATEMENT LABEL A
      THRØUGH S, FØR VALUES ØF A = B,C,D
          TRANSFER TØ A
B         PRINT CØMMENT $ FIRST PASS $
          TRANSFER TØ S
C         PRINT CØMMENT $ SECØND PASS $
          TRANSFER TØ S
D         PRINT CØMMENT $ THIRD PASS $
S     CØNTINUE
      PRINT CØMMENT $ NEXT-STATEMENT $
```

Note that the name A must be declared to be in statement label mode, since it does not appear as the label of any statement (i.e. in columns 1 through 10). The results of this statement would be just the printing of the four comments in order. It should be noted that the first three statements could be replaced with

```
INTEGER I
THRØUGH S, FØR VALUES ØF I = 0,1,2
    TRANSFER TØ A(I)
```

and the labels B, C and D replaced by A(0), A(1) and A(2) respectively. This iteration statement would produce the same results as the previous one.

Special precautions must be taken when the iteration variable is in function-name mode; these will be discussed in Chapter Seven.

We will now present further examples of iteration statements. One such loop is illustrated in the solution of the following problem:

> An integer n and a set of n floating point mode numbers, $a_0 \ldots a_{n-1}$, are read as data. Find and print out the subscript of the algebraically greatest of these numbers. The smallest such subscript should be chosen whenever this number is not unique.

A flowchart and the corresponding MAD routine are shown in Figure 6.8. This loop has a single index, i, which, during the execution of the statements in the scope of the iteration statement, is just equal to the number of the current pass through the loop (e.g. i = 3 on the third time through the loop). The loop is traversed for values of i running from 1 through n-1 (a total of n-1 times) and i equals n upon exit from the loop. At the end of the i-th pass through the loop, g is the subscript of the largest element of $a_0$ through $a_i$. If, on the i-th pass $a_i = a_g$, the Boolean expression has the value <u>false</u>, thus leaving g as the smaller subscript when two numbers are equally the largest. Before the first pass, g = 0, since $a_0$ is the largest and only element so far considered. The variable g must be initialized before entering the loop. In this MAD routine, the iteration variable is the index

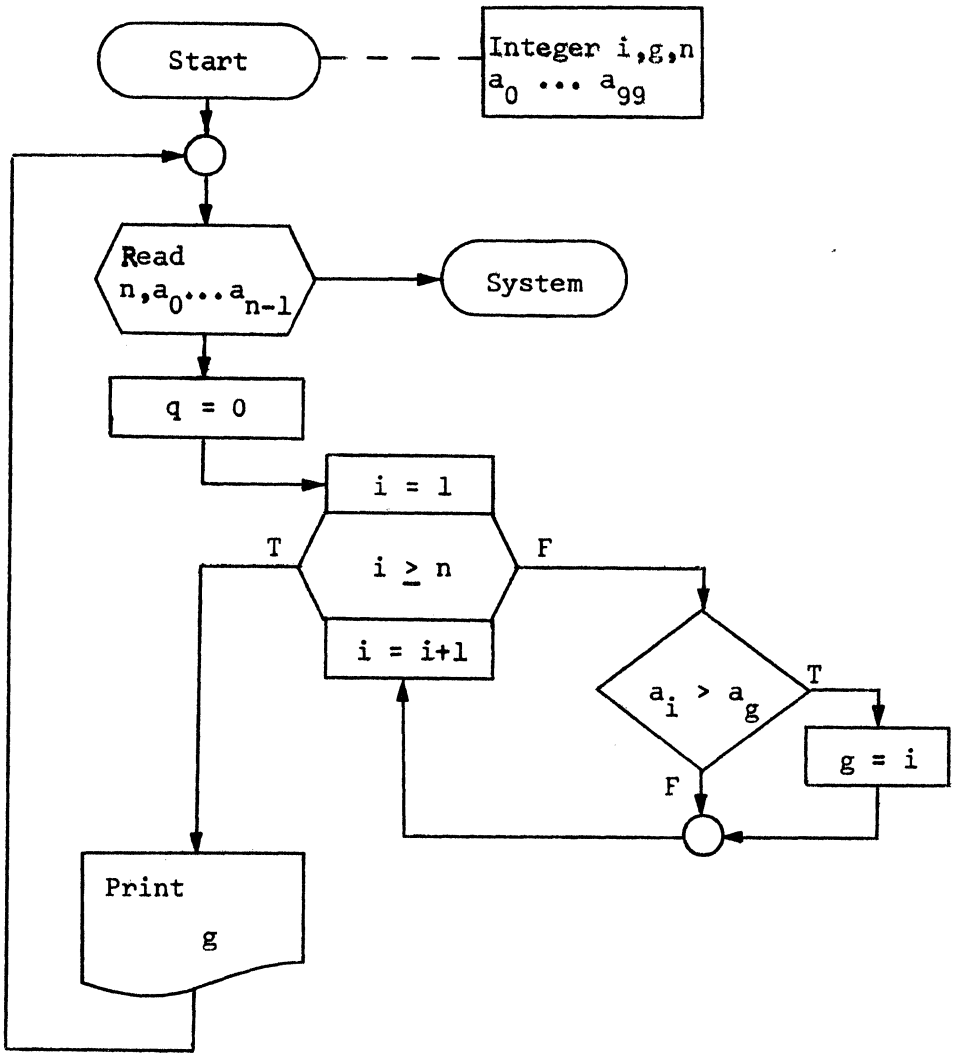



Figure 6.8a

```
          RRØUTINE TØ FIND THE SUBSCRIPT ØF THE LARGEST ELEMENT
          R    ØF A VECTØR A.
           INTEGER I, N, GRTST
           DIMENSIØN A(99)
START      READ AND PRINT DATA          N, A(0)...A(N-1)
           GRTST = 0
           THRØUGH S, FØR I = 1, 1, I.GE.N
               WHENEVER A(I) .G. A(GRTST), GRTST = I
S          CØNTINUE
           PRINT RESULTS GRTST
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 6.8b

of the loop. Most loops that utilize an index are implemented in this way. The label S could have been placed on the conditional statement and the CØNTINUE omitted.

Let us now consider a slight modification of this algorithm that results in a routine in which the CØNTINUE is required. The relation in the conditional could be reversed, changing the test in the flowchart to $a_i \leq a_g$ and consequently reversing the labels T and F on the decision box. The iteration statement in the modified routine would then become

```
      THRØUGH S, FØR I = 1, 1, I.GE.N
          WHENEVER A(I) .LE. A(GRTST), TRANSFER TØ S
          GRTST = I
S     CØNTINUE
```

The transfer to S causes the execution of the CØNTINUE statement, followed by the incrementing of the iteration parameter. The CØNTINUE could not be omitted in this arrangement, since there would then be no way to avoid executing GRTST = I. Note that a transfer to the THRØUGH statement is not a solution to this problem, since it would result in the initialization of I (i.e. setting I to 1) and would create an infinite loop. Although this structure would not normally be used for this problem, the need for transfers to the end of the iteration statement to increment the controlled variable without further action is quite common and other examples will appear in later sections.

If only the value of the largest component of the vector were required and not the subscript, the computational portion of the routine could be modified as follows:

```
      LRGST = A(0)
      THRØUGH S, FØR I = 1, 1, I.GE.N
          WHENEVER A(I) .G. LRGST, LRGST = A(I)
S     CØNTINUE
```

where GRTST is not used and LRGST is in floating point mode. Note that LRGST must be initialized to A(0), not to 0. Otherwise, the routine would give an answer of 0. if each element of A were negative (since 0. is larger than any negative number). Also, the relation evaluated in the conditional is essentially the same for this loop and that of Figure 6.8b, since LRGST and A(GRTST) are identical in value at the corresponding points in the execution of each routine.

An example of a search is found in the following problem:

> An integer n, a set of n floating point mode numbers, $b_1 \ldots b_n$, and the floating point mode number q are read as data. Find and print out the subscript of the first element of the vector b that is within .01% of q. If none are that close, an appropriate comment should be printed.

An algorithm for the solution of this problem is given in the flowchart and the corresponding MAD routine of Figure 6.9. This algorithm differs from the preceding one in a number of significant ways. First, although n numbers are being read in, the subscripts run from 1 to n, rather than from 0 to n-1. Thus, the initial value of the index, f, is 1 and the terminating condition is f>n. Also, in order to allow the computation on 100 components of b, space must be reserved for 101 elements (subscripts 0 through 100). Although this procedure wastes one storage space, it is commonly used, since it is often convenient to use one as the first subscript. This loop may be terminated either upon the unsuccessful completion of the search (after having examined all n components of b) or upon finding an element satisfying the requirement of the search. Note that the variable p was introduced, the value of which is .0001q, to avoid the necessity of executing that multiplication each time through the loop.

Since two floating point mode numbers are equal only if they are identical, two numbers that differ only in the 27th significant bit (approximately the 8th decimal digit) will produce the result <u>false</u> if they are compared using the relation "equal to." Furthermore, the computer representation of floating point numbers is not always exact and small errors occur in calculations due to roundoff. Thus, a test for approximate equality, such as the one used in this example, is the only meaningful one in many cases.

This routine can be modified by replacing the simple conditional in the scope of the loop, as well as the statement labeled FØUND and the one following it, with the following compound conditional:

```
WHENEVER .ABS.(B(F)-Q) .LE. P
     PRINT RESULTS F, B(F)
     TRANSFER TØ START
END ØF CØNDITIØNAL
```

In this case, the label S may be placed on the END ØF CØNDITIØNAL without any confusion, since its function is similar to that of a CØNTINUE statement.

Many algorithms involve the evaluation of polynomials of the form

$$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1} + a_nx^n$$

for given values of the $a_i$ and of x. If the algorithm is designed to work for more than one value of n, i.e. for different order polynomials, loops are normally employed. A straightforward approach, but one that is <u>quite inefficient</u>, would be to calculate and add in one term of the sum each time through the loop. Thus,

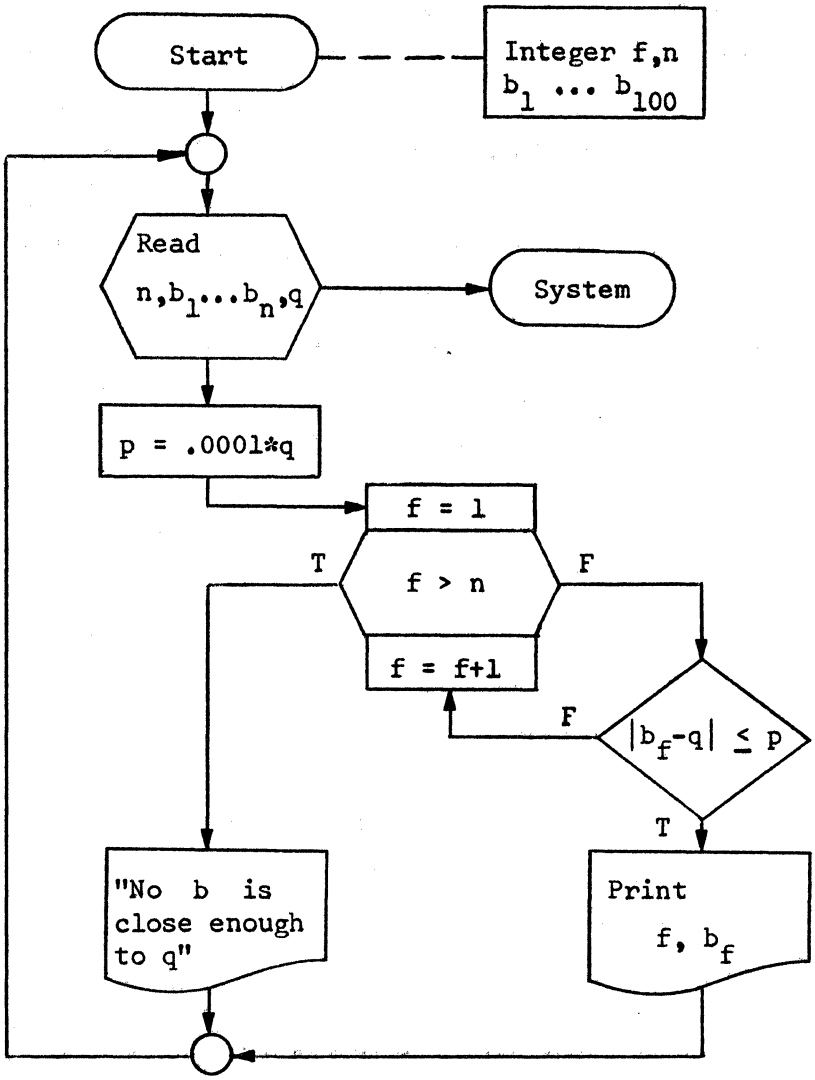



Figure 6.9a

```
            RRØUTINE TØ FIND THE FIRST ELEMENT ØF B WITHIN
            R    .01 PERCENT ØF Q
             INTEGER F, N
             DIMENSIØN B(100)
START        READ AND PRINT DATA      N, B(1)...B(N), Q
             P = .0001*Q
             THRØUGH S, FØR F = 1, 1, F.G.N
                 WHENEVER .ABS.(B(F)-Q) .LE. P, TRANSFER TØ FØUND
S            CØNTINUE
             PRINT CØMMENT $ NØ B IS CLØSE ENØUGH TØ Q. $
             TRANSFER TØ START
FØUND        PRINT RESULTS F, B(F)
             TRANSFER TØ START
             END ØF PRØGRAM
```

Figure 6.9b

statements such as

```
                INTEGER I, N
                P = A(0)
                THRØUGH S, FØR I = 1, 1, I.G.N
                    P = P + A(I)*X.P.I
S               CØNTINUE
```

would calculate the value of p for a given value of x. Note that on the i-th pass through the loop, one addition and i multiplications are required (since $x^i$ is calculated by (i-1) multiplications). Thus a total of $\frac{n(n+1)}{2}$ multiplications and n additions are needed. Furthermore, a subroutine must be called to calculate the power each time through the loop. That this is excessive is obvious when one realizes that $x^i$ is calculated each time without making use of the calculation of $x^{i-1}$ during the previous pass through the loop. A modification that takes advantage of the earlier calculations is as follows

```
                Y = X
                P = A(0)
                THRØUGH S, FØR I = 1, 1, I.G.N
                    P = P + A(I)*Y
                    Y = Y*X
S               CØNTINUE
```

This reduces the number of multiplications to 2n, eliminates the subroutine calls, and leaves the number of additions unchanged, resulting in a significant improvement over the first method. Finally, consider factoring p(x) in the following manner

$$a_0 + x(a_1 + \ldots + x(a_{n-2} + x(a_{n-1} + xa_n)) \ldots)$$

Multiplying this out clearly gives the same expression for p(x) as before. It should be noted that this is precisely the remainder term obtained when dividing p(y) by (y-x) and the calculations are identical to those done when evaluating this polynomial by synthetic division. The corresponding MAD statements are as follows:

```
                P = A(N)
                THRØUGH S, FØR I = N-1, -1, I.L.0
                    P = P*X + A(I)
S               CØNTINUE
```

This approach requires only one multiplication and one addition each time around the loop and is more efficient than either of the other methods for all polynomials. The advantage of this method is greatest for higher order polynomials. Even for tenth order polynomials, the first method requires 55 multiplications and the second 20 multiplications, whereas this one requires only 10 multiplications. A major difference between this loop and those previously discussed is that in this case the index is initialized at its maximum value and is incremented by a negative amount. There are many situations in which that is the most convenient and efficient

approach.

Examples of iterations will be included in the next two sections. Some of the algorithms presented there will illustrate loops that do not use an index and iteration statements in which the iteration variable is incremented by a different amount each time through the loop. Other examples of the use of loops will appear throughout the remainder of the text.

## Exercises

1. Describe what happens as a result of each of the following sets of statements. If the operation depends upon the value of a variable, indicate what happens for all possible values of that variable. Assume that the normal mode is integer in all cases.

a)
```
      S       THRØUGH S, FØR I = 1, 0, I.GE.A
```

b)
```
              THRØUGH T, FØR J = 0, 1, J.G.5
                  PRINT RESULTS J
                  J = J + 1
                  PRINT RESULTS J
      T       CØNTINUE
```

c)
```
              THRØUGH S, FØR VALUES ØF I = 0, 1, 2, 3, 4, 5
                  PRINT RESULTS I
                  I = I + 1
                  PRINT RESULTS I
      S       CØNTINUE
```

d)
```
              THRØUGH LØØP, FØR I = 1, I, I.E.1K8
                  PRINT RESULTS I
      LØØP    CØNTINUE
```

e)
```
              THRØUGH Q, FØR P = 1, 10, P.E.10000
      Q       CØNTINUE
              PRINT CØMMENT $ HØW DID I GET ØUT ØF THERE $
```

f)
```
              SUM = 0
              THRØUGH S, FØR K = 0, 2, K.L.5
                  SUM = SUM + K
      S       CØNTINUE
```

g)
```
              I = 3
              THRØUGH T, FØR J = 0, I, J.E.I
                  I = I - 1
                  PRINT RESULTS I, J
      T       CØNTINUE
              PRINT RESULTS I
```

h)
```
      P       THRØUGH T, FØR I = 0, 1, I.G.10
                  PRINT CØMMENT $ THIS IS A LØØP. $
      T       TRANSFER TØ P
```

```
i)              THRØUGH J, FØR I = 1, X.P.4-1, I.P.3.G.10
                    X = I
      J         CØNTINUE

j)              FLØATING PØINT I, A(5)
                J = 0
                THRØUGH X, FØR I = 0., A(J), J.G.4
                    J = J + 1
      X         CØNTINUE
                PRINT RESULTS I/5.

k)              READ AND PRINT DATA     D
                VECTØR VALUES A = .01, .02, .025, .03, .05
                THRØUGH S, FØR I = 0, 1, I.G.4
                    TAX = D*A(I)
                    PRINT RESULTS TAX
      S         CØNTINUE
                FLØATING PØINT   TAX, D

l)              FLØATING PØINT X
      S         THRØUGH S, FØR I = 0, 1, I+1 .G. X
                    PRINT RESULTS X, I

m)              I = 0.
                STATEMENT LABEL A
                THRØUGH S, FØR VALUES ØF A = B, C, B, D
                    I = I + 1.
                    TRANSFER TØ A
      B             I = I + 2.
                    PRINT RESULTS I
                    TRANSFER TØ S
      C             PRINT RESULTS I
      S         CØNTINUE
      D         PRINT RESULTS I
```

2. For the following polynomial

$$p(x) = 4.6x^5 + 2.1x^4 - 5.7x^3 + 6.1x^2 - 3.9x + 7.6$$

   a) Evaluate, by hand, p(2.5) using each of the three methods described in this section.

   b) Divide p(x) by x - 2.5 (either using long division or synthetic division). Compare the remainder with the answer to part a.

3. Repeat Exercise 4, Page 33 using the iteration statement.

4. Draw a flowchart and write a MAD main routine to calculate the subscript of that component of the vector a (elements $a_1 \ldots a_n$) having the smallest absolute value. If two or more elements have the same value and are smallest, print the largest of their subscripts.

5. The following MAD main routine, with the accompanying data sets, contains many errors and bad usages. It was intended to calculate and print out n evenly spaced values of a vector X, where n and the smallest and largest values of X are given as data. The data include N, X1, and XN and the result is to be a vector XVEC(1)...XVEC(N). For example, if the data set was

N = 4, X1 = 1.5, XN = 2.6*

the output should be

XVEC(1)...XVEC(4)

1.500000 1.866667 2.233333 2.600000

List and explain how to correct the following:

a) All syntactic errors, i.e. those errors that will prevent the routine from compiling.

b) All errors in data.

c) All logical errors, i.e. those errors that will prevent the routine from getting the correct answer.

d) All bad usages, i.e. those things that do not fit in the categories a, b, or c, but do cause the machine more work or lead to confusion on the part of the reader.

```
STAPH       INTEGER N
            DIMENSIØN XVEC(N)
            READ AND PRINT DATA
            XVEC(1) = X1
            DETAX = (X(N)-X(1))/N)
            THRØUGH JIM FØR I = 1, 1, I = N + 1
JIM         XVEC(I-1) + DELTAX = XVEC(I)
            TRANSFER TØ START
            PRINT RESULTS XVEC
            END ØF PRØGRAM
$DATA
N = 3, X1 = 2., XN = 5.
N = 6, X1 = 6.33, X6 = 8.124*
N = 100, X1 = 2000, XN = 2000.001*
N = 5, XN = 5., X1 = 5.*
N = 1, X1 = 5., XN = 6.*
```

## 6.4 Iterative Solutions for the Zeros of Functions

In this section and the next section, several approaches to the problem of finding a real zero of a single valued function f of one variable x (i.e., a value of x such that the function f(x) is equal to zero) are discussed. Except for special cases, the problem must be attacked by successive approximations. In this section, the half interval method is discussed in detail and some other methods that do not require the calculus are presented. Section 6.5 is devoted to the Newton-Raphson method (sometimes referred to as Newton's method), which does utilize the calculus. Methods for finding complex zeros of functions do exist, but are beyond the scope of the text.[2]

Since it may not be possible to express the values of the zeros exactly in a computer, we must usually be satisfied with an approximate solution to the problem. Even if an exact solution can be obtained, we may not need that much precision or we may not be willing to spend the resources required to obtain it. The accuracy which can be obtained is limited primarily by the amount of effort which one is willing to devote to the work. This effort involves the work of people in analysing the problem and setting up the computation as well as the computer time and memory used.

One iterative technique, often referred to as linear iteration, is based on the derivation from the equation $f(x) = 0$ of an equivalent equation of the form

$$x = g(x)$$

where g is dependent on x except in trivial circumstances. In general, many such equivalent equations can be found. As an example, consider

$$f(x) = x^2 - 5x + 6$$

in which case

$$g_1(x) = 5 - \frac{6}{x}$$

or

$$g_2(x) = .2(x^2 + 6)$$

might be obtained. The iterative technique involves making an initial approximation, $x_0$, which is used in evaluating g(x) to obtain a next (and hopefully better) approximation to the zero, and then successively repeating this process. The formula

$$x_i = g(x_{i-1}) \qquad i = 1,2,3,\ldots$$

---

[2] Conte, S. D., Elementary Numerical Analysis--An Algorithmic Approach, McGraw-Hill, 1965, New York, Chapter 2.

gives each approximation in terms of the previous one.

One problem with this approach is that often it does not converge, i.e. the successive $x_i$ do not approach a zero of f(x). Even if the iterations do converge toward a zero, that convergence is often quite slow, requiring a large value of i and thus a great deal of work before $x_i$ gets very close to a zero of f(x).

In the example above, an iteration using $g_2$ converges to a solution only if $|x_0| \leq 3$, whereas one using $g_1$ converges except when some $x_i$ equals zero. (This may occur when $x_0$ = 0., 1.2 or certain values between 1.5 and 2.). The first 25 approximations for $x_0$ = 1, using each of the two iterations is shown in Table 6.1. As can be seen from this table, even 25 iterations does not give an answer closer than one part in 40,000 (for $g_1$; only 1 part in 1250 for $g_2$) whereas the machine has precision to better than one part in 100 million. Because this method generally converges slowly, if it converges at all, the derivation of a function g which will provide rapid convergence and the selection of an appropriate starting point should be done carefully[3]. Other methods often converge more rapidly.

Two methods for the iterative solution for a zero of a function are the half interval method and regula falsi. Each method guarantees a solution for the approximation to the zero under the following conditions:

1) f(x) is a real, single-valued, function of a single real variable, x.
2) Two points, x = a and x = b (where a<b), exist such that f(a) and f(b) are not both positive or both negative.
3) f(x) is continuous (and thus finite) in the closed interval $a \leq x \leq b$.

If these conditions are satisfied, then there is always at least one zero of f(x) in the interval $a \leq x \leq b$. Either of these methods will always converge to one of the zeros in the interval. We will develop in this section an algorithm for the half interval method and then indicate the modifications necessary for regula falsi.

Before discussing the appropriate terminating conditions, we will present the general approach and, thus, defer a definition of when the approximation is "close enough" until later. A typical function f(x) is shown in Figure 6.10. In this example, the value of the function at x = a, f(a), is positive and at x = b, f(b), is negative. We will denote the size of the interval, b-a, by h. The first step in the procedure is to choose a point half way between a and b, denoted by c in

[3]Henrici, Peter, Elements of Numerical Analysis, John Wiley, 1964, New York, Chapter 4.

Table 6.1 Iterative Solution of $x^2 - 5x + 6$

| i | $x_i$ using $x_0 = 1$ and | |
|---|---|---|
| | $g_1(x) = 5 - \frac{6}{x}$ | $g_2(x) = .2(x^2+6)$ |
| 0 | 1.000000 | 1.000000 |
| 1 | -1.000000 | 1.400000 |
| 2 | 11.000000 | 1.592000 |
| 3 | 4.454545 | 1.706893 |
| 4 | 3.653061 | 1.782697 |
| 5 | 3.357542 | 1.835601 |
| 6 | 3.212978 | 1.873886 |
| 7 | 3.132574 | 1.902290 |
| 8 | 3.084642 | 1.923741 |
| 9 | 3.054880 | 1.940156 |
| 10 | 3.035929 | 1.952841 |
| 11 | 3.023669 | 1.962718 |
| 12 | 3.015656 | 1.970452 |
| 13 | 3.010383 | 1.976536 |
| 14 | 3.006898 | 1.981339 |
| 15 | 3.004588 | 1.985141 |
| 16 | 3.003054 | 1.988157 |
| 17 | 3.002034 | 1.990554 |
| 18 | 3.001355 | 1.992461 |
| 19 | 3.000903 | 1.993980 |
| 20 | 3.000602 | 1.995191 |
| 21 | 3.000401 | 1.996157 |
| 22 | 3.000267 | 1.996929 |
| 23 | 3.000178 | 1.997545 |
| 24 | 3.000119 | 1.998037 |
| 25 | 3.000079 | 1.998430 |

Figure 6.10. The point c divides the interval into two equal parts, namely

$$a \leq x \leq c$$

$$c \leq x \leq b$$

Each of these intervals is one half the size of the original interval, i.e. their size is $\frac{h}{2}$. There are three possibilities at this point. First, c might be close enough to the zero, in which case the iteration may be terminated. Second, f(c) and f(a) may be of opposite sign (and consequently f(c) and f(b) are of the same sign), in which case there is at least one zero in the interval

$$a \leq x \leq c$$

This is the case in the example of Figure 6.10. Finally, f(c) and f(b) may be of opposite sign, in which case there is a zero in the interval

$$c \leq x \leq b.$$

In either of the last two cases, we are left with the problem of finding a zero of f(x) in a new interval, half the size of the original interval. By repeating this process, the size of the interval is cut in half on each iteration. A flowchart

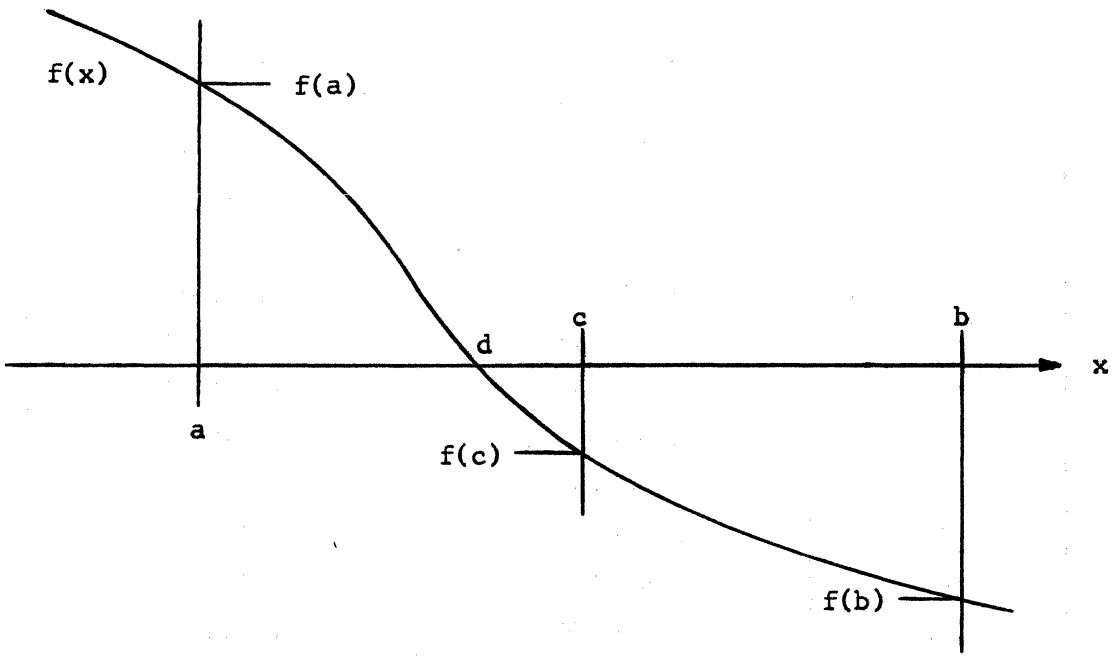



Figure 6.10

showing this process is given in Figure 6.11.

Before presenting a more detailed flowchart for this algorithm, we will first consider the appropriate terminating conditions for this iteration. Two common terminating conditions are:

1) $h \leq \delta$, i.e. the interval is small enough, or $f(c) = 0$.
2) $|f(c)| \leq \varepsilon$, i.e. the value of the function at the midpoint is small enough.

In the first case, there is always a zero in the interval h, and, if the midpoint of that is chosen as the approximation to that root, then that approximation is within $\frac{\delta}{2}$ of the actual zero. This terminating condition has the disadvantage that even if the first approximation to the zero is very close (possibly much closer than $\frac{\delta}{2}$), the iteration continues until the interval is reduced to less than $\delta$ unless f(c)

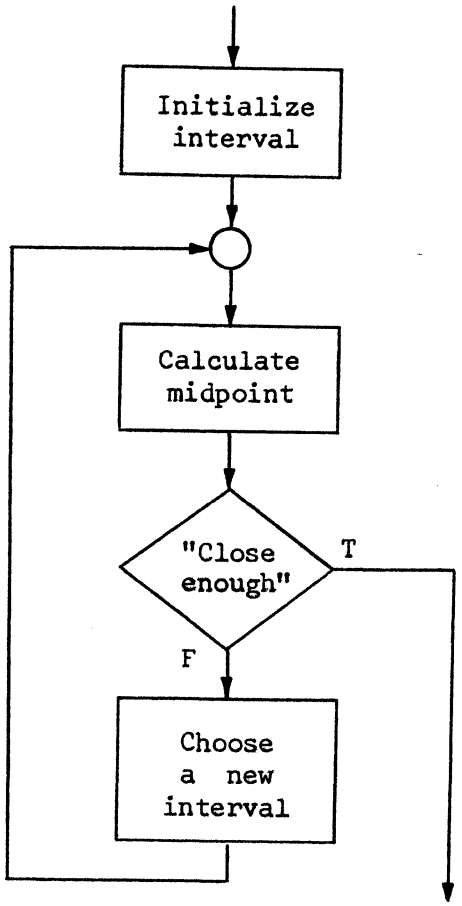



Figure 6.11

is exactly zero (to machine precision). Thus, although the terminating condition does not involve the number of times through the loop, it is possible to calculate that number before entering the loop. Since, on the i-th pass through the loop, the interval is given by $(b-a)/2^{i-1}$, then the number of times the loop is traversed, n, is given by the solution of

$$\frac{b-a}{2^{n-1}} \leq \delta$$

or

$$n \geq \log_2 \left(\frac{b-a}{\delta}\right) + 1$$

where the solution desired is the first integer at least as great as the expression on the right.

The use of the second terminating condition normally eliminates this difficulty, since, if c is close to a root, f(c) is close to zero. This condition is illustrated graphically in Figure 6.12. Whenever the function is in the shaded region, the

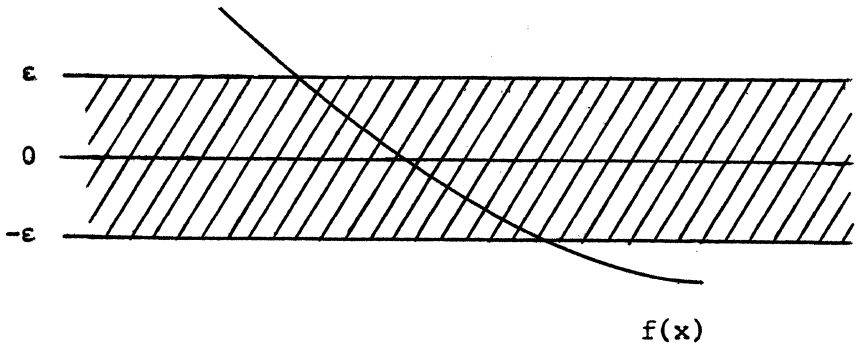



Figure 6.12

condition $|f(x)| \leq \varepsilon$ is satisfied. Although this condition does not guarantee that x is close to the root, it normally does produce satisfactory results if ε is chosen small enough. Two situations in which this is not satisfactory are shown in Figure 6.13. In the first case, the curve approaches the axis at one point (such that $|f(x)| < \varepsilon$) but there is no root near that point. An example of such a function is $f(x) = x^3 - 4x^2 + 5.01x - 2.02$ which behaves that way in the vicinity of x = 1. In the second case, the function is close to zero over a wide range of x. In that case, an approximation satisfying this terminating condition might still be quite far from the root. Consider, for example,
$f(x) = x^6 + 7x^5 + 20x^4 + 30x^3 + 25x^2 + 11x + 2$ in the region of $-1.5 \leq x \leq -.5$.

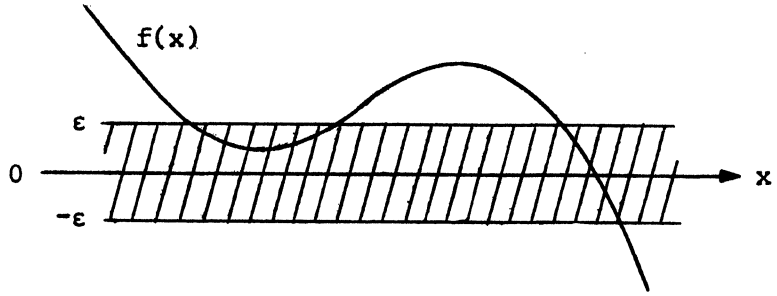



Figure 6.13a

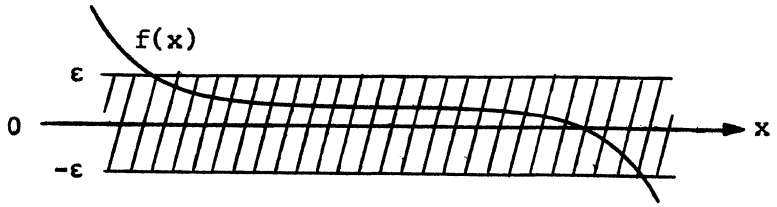



Figure 6.13b

Before developing a complete algorithm for this problem, let us first consider a numerical example. Table 6.2 shows the interval, the midpoint and the value of the function at the midpoint for the solution of $f(x) = x^2 - 2.4x + .44$ in the interval $0 \leq x \leq 2$. The end points of the interval at each iteration are denoted by q and r and the midpoint by c. This example was chosen so that the reader can follow the computations; one would <u>not</u> utilize this method to solve a quadratic. It should be noted that either the interval, h, or the magnitude of the function at the midpoint of the interval, f(c), could be used as the terminating condition with similar results. On the other hand, the function $f(x) = x^2 - 3.51x + 2.505$, in the same interval, has a root at x = 1.002 which could be found within a tolerance of .002 on the first interval, i.e. c = 1.000 and f(c) = .0031. Thus, if the terminating condition were $|f(c)| \leq \varepsilon$ and $\varepsilon$=.005, the iteration would require only a single pass. On the other hand, if the terminating condition was $h \leq \delta$ where $\delta$ = .005, ten iterations would be required.

Table 6.2 Half Interval Solution of $f(x) = x^2 - 2.4x + .44$

| q | c | r | h | f(c) |
|---|---|---|---|---|
| .000000 | 1.000000 | 2.000000 | 2.000000 | -.960000 |
| .000000 | .500000 | 1.000000 | 1.000000 | -.510000 |
| .000000 | .250000 | .500000 | .500000 | -.097500 |
| .000000 | .125000 | .250000 | .250000 | .155625 |
| .125000 | .187500 | .250000 | .125000 | .025156 |
| .187500 | .218750 | .250000 | .062500 | -.037148 |
| .187500 | .203125 | .218750 | .031250 | -.006240 |
| .187500 | .195313 | .203125 | .015625 | .009397 |
| .195313 | .199219 | .203125 | .007813 | .001563 |
| .199219 | .201172 | .203125 | .003906 | -.002342 |
| .199219 | .200195 | .201172 | .001953 | -.000391 |
| .199219 | .199707 | .200195 | .000977 | -.000586 |
| .199707 | .199951 | .200195 | .000488 | .000098 |
| a = .000000 | | | b = 2.000000 | |
| f(a) = .440000 | | | f(b) = -.360000 | |

Let us now consider an algorithm suitable for implementation on a computer. Using $|f(x)| \leq \varepsilon$ as the terminating condition, the input data must include

1) the function[4], f(x)
2) the interval, namely a and b
3) the value of $\varepsilon$

This algorithm, like most general algorithms designed for computer application, should have two additional characteristics. First, the values of the inputs should not be modified during the execution of the routine (e.g. a and b are unchanged) since these values are often required by other algorithms dependent upon this one.

[4]The word data is used here in the general sense. The function to be used must somehow be specified. The use of a function as an input to a subroutine will be discussed in Chapter Seven.

Second, at least the most common type of data for which the algorithm will not work must be checked for. To implement the first of these, we must use names other than a and b for the bounds of the sub-intervals (i.e. those intervals after the first iteration). The algorithm should check that the values of the function at the endpoints are of opposite sign[5]. Although the problem statement specifies that a < b, it can be seen that this is an unnecessary restriction, i.e. the algorithm is also valid for a > b.

The algorithm can be divided into two sections. The first part, including the tests for special conditions and the initializing of the interval is shown in Figure 6.14. Since the values of f(a) and f(b) are required in several places in this portion of the algorithm, these are calculated only once and a name is assigned to each. No further calculations are required if either a or b satisfy the terminating condition. If the endpoints of the interval are not roots, then the signs of the endpoints are checked. In order to avoid the possibility of overflow or underflow s is set equal to +1. if f(a) > 0. and -1. if f(a) < 0. This is possible since only the sign of f(a) is used and, thus, s*f(b) has the same sign as f(a)*f(b), which is negative only when the signs differ.

The flowchart for the iteration portion of the algorithm is shown in Figure 6.15. It should be noted that this flowchart is not in a form compatible with the MAD iteration statement. Although it can be forced into that form, that would be somewhat artificial. Because of the finite number of significant digits in the computer, it is necessary to add some steps not included in the general flowchart. It must be determined that the midpoint is actually distinct from the endpoint it is replacing. This would not be true only if the two ends of the interval were so close that a distinct midpoint could not be found, but the condition $|f(c)| \leq \varepsilon$ was not satisfied. Although this would be unusual, $\varepsilon$ could be chosen that small and without this precaution, an infinite loop would ensue. In this algorithm, if such a condition results, that point is accepted as a solution, since the machine prevents any closer determination of the root. Furthermore, the maximum error in this case is much less than $10^{-7}$. It should be noted that if f(a) (i.e. the value of the function at the left end of the original interval) is negative (positive), f(q) remains negative (positive). Similarly, the sign of the function at the right end of

[5] If f(a) and f(b) have the same sign, but it is still desired to obtain a root in the interval $a \leq x \leq b$, then one may search for a point, d, in this interval such that the sign of f(d) differs from that of f(a) and f(b). If such a point is found, then each of the intervals $a \leq x \leq d$ and $d \leq x \leq b$ contain a root of f(x). A systematic method of searching is suggested in Exercise 5 at the end of this section.

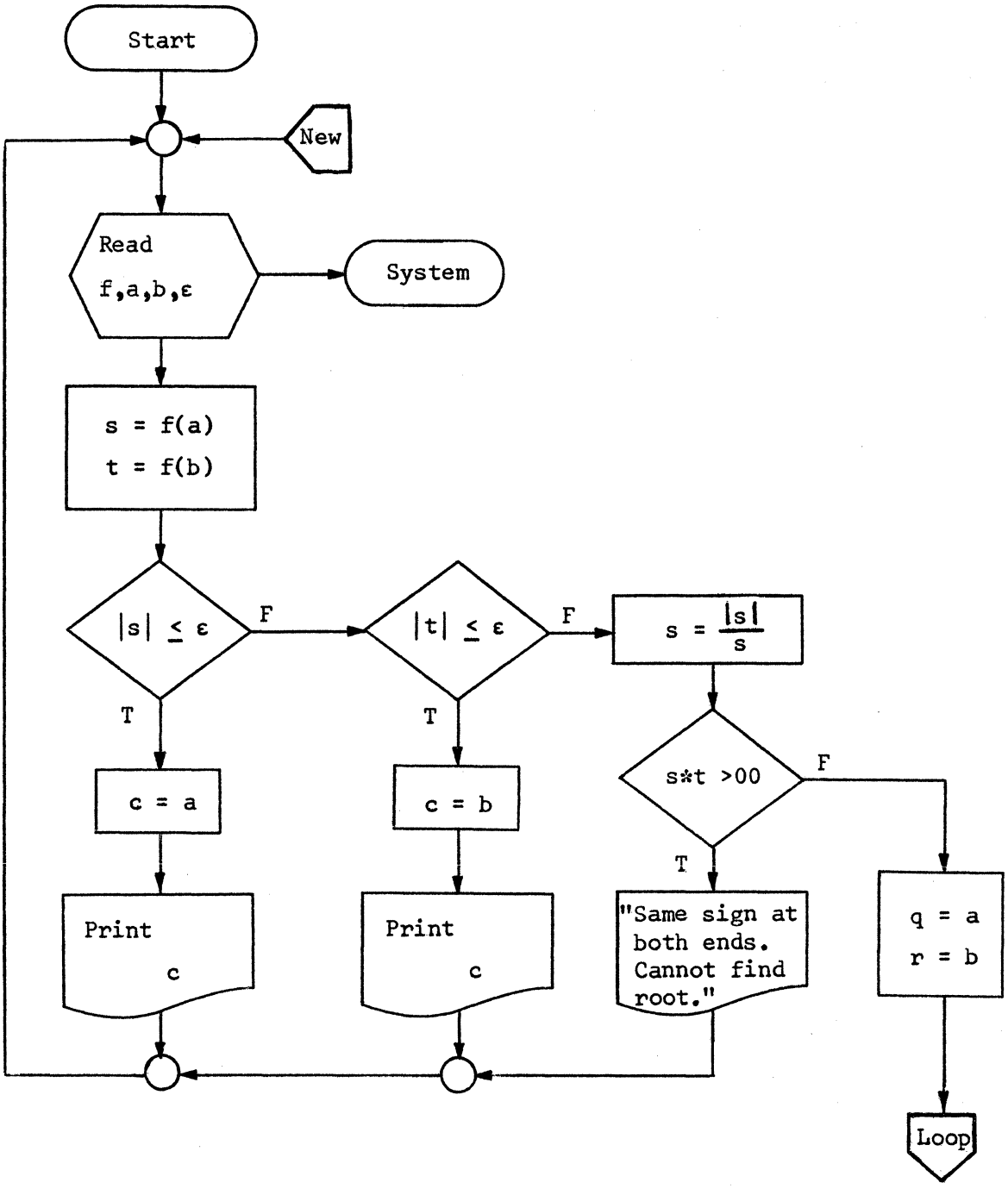

Start
New
Read
f,a,b,ε
System
s = f(a)
t = f(b)
|s| ≤ ε
F
|t| ≤ ε
F
s = |s| / s
T
T
s*t >00
F
c = a
c = b
T
Print
c
Print
c
"Same sign at both ends. Cannot find root."
q = a
r = b
Loop

Figure 6.14

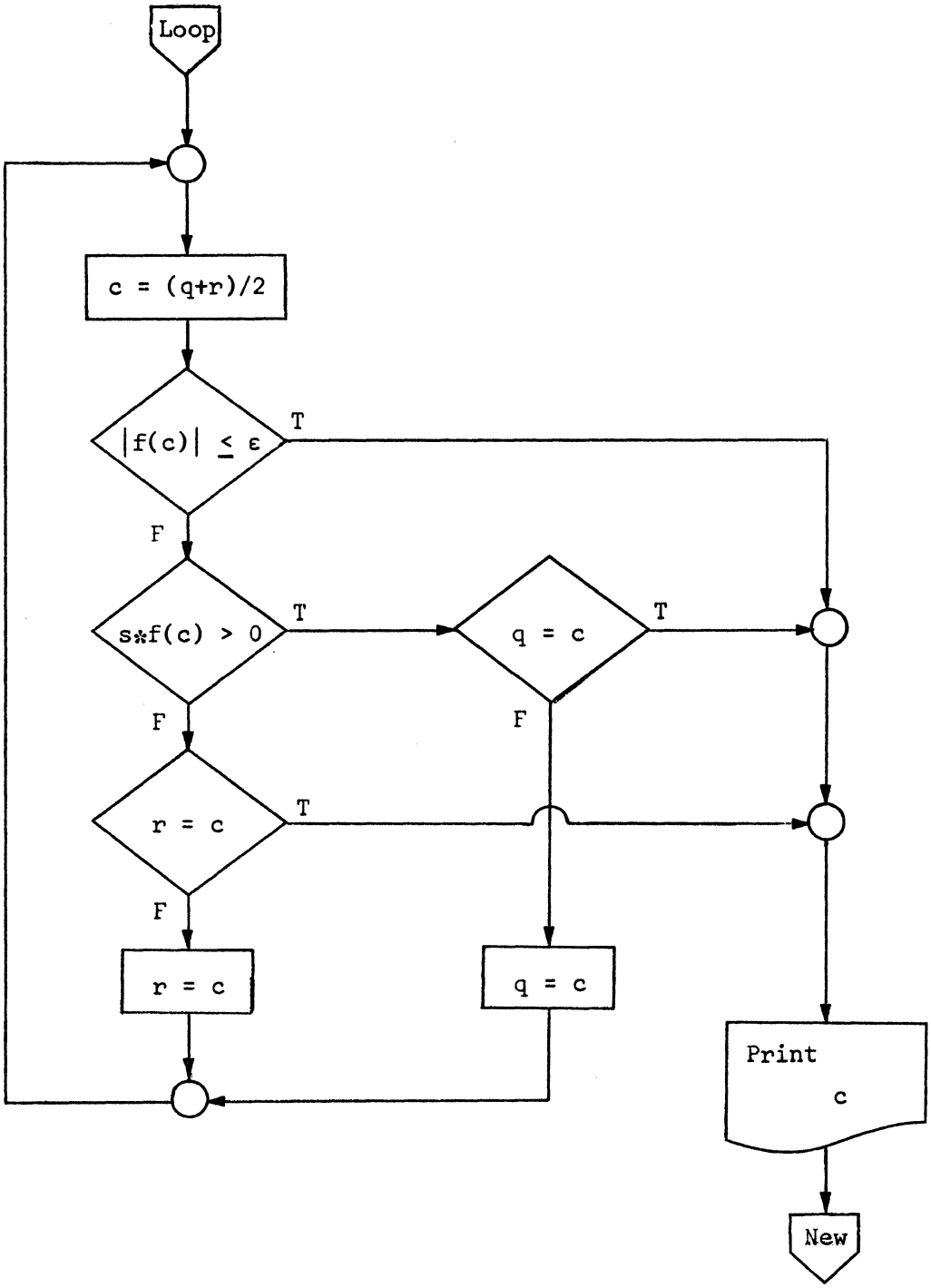



Figure 6.15

the intervals does not change. Thus, the sign of s*f(c) is sufficient to determine which end of the interval is replaced by the point c.

The method of regula falsi differs from the half interval method only in the method by which the point c is chosen. Figure 6.16 illustrates this approach. The point c is chosen as the intersection of the x axis and a straight line connecting the point (a, f(a)) and the point (b, f(b)). In many cases, this method converges

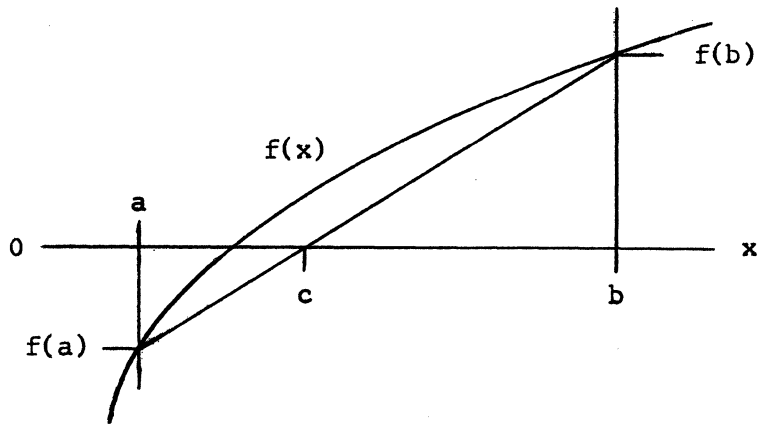



Figure 6.16

in fewer iterations than does the half-interval method. However, each iteration requires more calculation than does the half-interval method and it is thus not possible to determine in general which method converges with the least total work.

One of the most common applications of iterative techniques in a computer is the calculation of the square root of a number. The library routine available in most computers involves an iteration. One such iterative technique, known as "averaging", is useful in the calculation of the square root. Averaging is based on the observation that if x is an approximation to the square root of a, such that $x \geq \sqrt{a}$, then

$$\frac{a}{x} \leq \sqrt{a} \leq x$$

with both equalities holding when $x = \sqrt{a}$. If the midpoint, $y = \frac{1}{2}(x + \frac{a}{x})$, is chosen as the next approximation, a new interval is defined, bounded by y and $\frac{a}{y}$, as is illustrated in Figure 6.17. It can be shown that the point y falls in the interval $\sqrt{a} < y < x$ and that $\frac{a}{y}$ is in the interval $\frac{a}{x} < \frac{a}{y} < \sqrt{a}$. Thus, the second interval con- the root and is less than half of the size of the first. Actually, it is not necessary that the first approximation be larger than $\sqrt{a}$. If x had been smaller than $\sqrt{a}$ then $\frac{a}{x}$ would be larger than $\sqrt{a}$ and their roles could simply be interchanged.

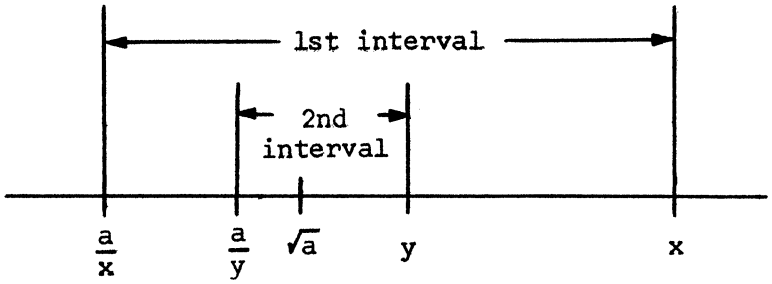



Figure 6.17

Repeating this process, the iteration formula

$$x_{i+1} = \frac{x_i^2 + a}{2x_i} = \tfrac{1}{2}(x_i + \frac{a}{x_i})$$

gives the next approximation in terms of the current one. As an example, if a = 2 and as the initial guess, $x_0$ is chosen as 2, then the next three successive approximations are 1.50000, 1.41667, and 1.41422, where the actual value of the square root is (to 6 decimal places) 1.414214. Thus, in only three iterations, an answer accurate to one part in 200,000 is obtained. A careful choice of the initial approximation may make it possible to use only a few (in some cases a fixed number such as 2 or 3) of these iterations to achieve the desired accuracy. One such method is suggested in Exercise 5, Section 6.5.

The terminating condition for this iteration is often that the magnitude of the change in the approximation becomes small, i.e. after i+1 iterations the test would be to see whether

$$|x_{i+1} - x_i| = x_i - x_{i+1} \leq \varepsilon$$

or, equivalently

$$x_i - \frac{x_i^2 + a}{2x_i} = \frac{x_i^2 - a}{2x_i} = \tfrac{1}{2}(x_i - \frac{a}{x_i}) \leq \varepsilon$$

Absolute values need not be taken since the differences used are always positive. This difference is usually computed first since after the termination test it can then be used to modify the i-th approximation to obtain the (i+1)st approximation. It should be noted that the square root of a is a zero of $f(x) = x^2 - a$ and so this terminating condition is also a measure of how close f(x) is to zero.

The same formula can also be obtained by the technique of linear iteration, which was the first method presented in this section. The equation $f(x) = 0$ can be solved for x, giving

$$x = \frac{a}{x}$$

Then adding x to both sides and dividing by 2, one obtains

$$x = \tfrac{1}{2}(x + \frac{a}{x}) = g(x).$$

The iteration formula

$$x_{i+1} = \tfrac{1}{2}(x_i + \frac{a}{x_i})$$

is thus obtained.

Finally, as we will see in the next section, the same iteration formula can be obtained for the square root by applying the Newton-Raphson iterative method.

Exercises

1. Use the half interval method to calculate, by hand, a root in each of the following cases. Show the interval after each iteration.

a) $f(x) = x^3 - 5x^2 - x + 5$, terminating condition: $h \leq .1$

i) a = -1 b = 0

ii) a = -32 b = 32

iii) a = 0 b = 2

iv) a = 0 b = 3

v) a = 1.5 b = 5.5

b) $f(x) = \sin x + .2$, a=0, b=1.5, terminating condition: $|f(x)| \leq .1$

2. Write a MAD main routine to implement the algorithm for the half interval method. Use an n-th order polynomial for f(x), where the coefficients, the order of the polynomial, the interval and $\varepsilon$ are read in as data.

3. Draw a flowchart and write the corresponding MAD routine to calculate the values shown in Table 6.1.

4. Draw a flowchart and write the corresponding MAD routine to calculate the square root of a floating point mode number, b (read as data), using averaging. The terminating condition should be that the change between the current approximation and the last is less than .01% of the current approximation, in magnitude. Choose as a starting point $\frac{a+1}{2}$, which it can be verified is always greater than the actual root.

5. One of the shortcomings of the half interval method is that it requires that the value of the function at the endpoints be of different sign (unless it is zero at one end). Modify the algorithm for the half interval method to make the following search if the value of the function is of the same sign at both endpoints. Examine the points (using d = b - a).

$$a + \frac{1}{2} d$$
$$a + \frac{1}{4} d$$
$$a + \frac{3}{4} d$$
$$a + \frac{1}{8} d$$
$$a + \frac{3}{8} d$$
$$a + \frac{5}{8} d$$
$$a + \frac{7}{8} d$$
$$a + \frac{1}{16} d$$
$$\vdots$$

or in general

$$a + \frac{2i - 1}{2^n} d$$

where n = 1, 2, 3, ... and i - 1, 2, ..., $2^{n-1}$ for each n. As soon as the function is of different sign from the endpoints at one of these points (or is zero), accept that and the nearest endpoint as q and r and begin the loop. If no point of different sign is found when $\frac{d}{2^n}$ becomes less than $\delta$, assume that no root can be found and print an appropriate message.

6. If the terminating condition of Exercise 4 is modified to read that the change between the current approximation and the next is less than .01% of the current approximation, in magnitude, then the entire iteration can be written in a single THRØUGH statement. Draw a flowchart and write the corresponding MAD routine in that way.

## 6.5 The Newton-Raphson Iterative Method

We will now develop another iterative method, known as the Newton-Raphson Method, for finding a real zero of a single-valued differentiable function. This method proceeds from an initial approximation of the zero through successive approximations until an approximation "close enough" to the zero is found. This method has two advantages over the half-interval method presented in the last section. First, an interval in which the root is known to exist is not required; only an initial approximation to the root is required. Although this approximation markedly affects the rapidity of convergence, and in some cases influences whether this procedure converges at all, it is often possible to begin at any arbitrary point and still converge to a root. We will discuss the choice of this starting point in detail below. Second, this method often converges very rapidly to the root, particularly if a good first approximation is made. Typical library routines use such an iteration to calculate the square root, and obtain 27 bits of accuracy (approximately 8 decimal digits) in only two iterations. This method has the disadvantage that, except in some special cases (e.g. a quadratic polynomial), convergence is not guaranteed unless a starting point is chosen close to the actual root. Furthermore, if the function has several zeros this method (if it converges) will find only one of these zeros, not necessarily the one closest to the starting point. This method also requires the ability to calculate the value of the derivative of f(x) at each of the approximations to the zero.

At a given approximation to the root, $x_i$, the Newton-Raphson method obtains the next approximation, $x_{i+1}$, by assuming that the function can be approximated by the straight line which is tangent to the curve f(x) at the point $(x_i, f(x_i))$. The slope of this line is equal to the derivative of the function evaluated at $x_i$, i.e. $f'(x_i)$. This is illustrated in Figure 6.18. The next approximation is then taken as the intersection of this tangent line and the x axis.

The formula for $x_{i+1}$ can be derived either from these geometrical considerations or from the truncated Taylor series approximation. First, from a geometrical point of view, $f'(x_i)$, the slope of the tangent, is given by

$$f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}}$$

Thus, solving for $x_{i+1}$, we obtain

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Second, if f(x) is expanded in a Taylor Series about $x_i$, the following is obtained:

$$f(x) = f(x_i) + (x-x_i)\, f'(x_i) + \frac{(x-x_i)^2}{2!} f''(x_i) + \ldots$$

Using only the first two terms of the series, and choosing $x_{i+1}$ as that value of x that reduces f(x) to zero, we obtain

$$\hat{f}(x_{i+1}) = 0 = f(x_i) + (x_{i+1}-x_i)\, f'(x_i)$$

where the $\hat{f}$ indicates an approximation of f. This reduces to the same formula as before, namely,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

As an example, consider

$$f(x) = x^2-b$$

The zeros of this function are the square roots of b. The derivative, f'(x), is 2x and thus the iteration formula is

$$x_{i+1} = x_i - \frac{x_i^2-b}{2x_i} = \frac{x_i^2+b}{2x_i} = 1/2\left[x_i+\frac{b}{x_i}\right]$$

which is the same formula produced by the method of averaging discussed in the last section.

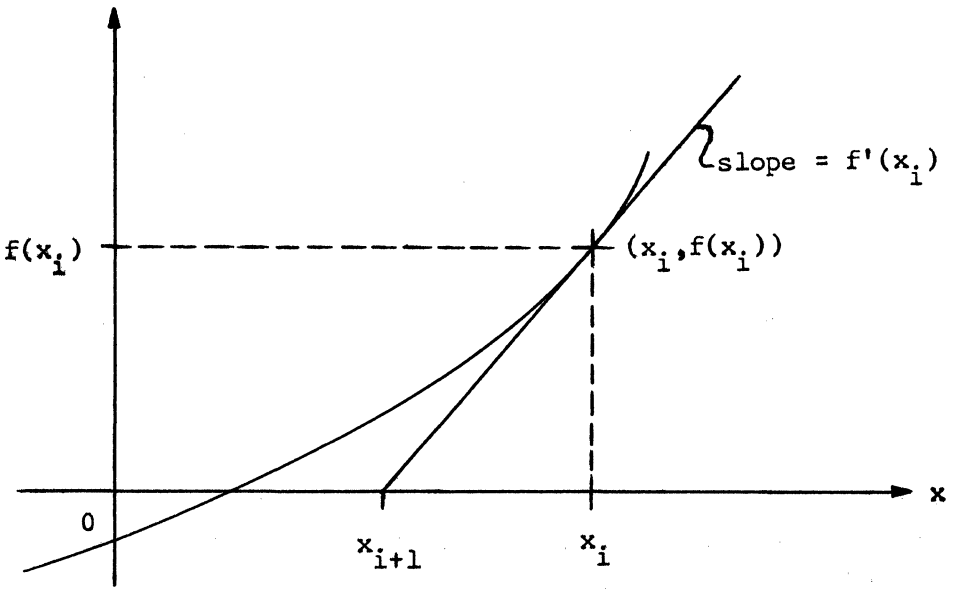



Figure 6.18

The Newton-Raphson Method does not always converge. In addition to the possibility that successive approximations wander, i.e. do not settle down on any zero, two more easily recognizable situations may occur that prevent convergence. First, the derivative, $f'(x)$, may be zero or very close to zero at one of the approximations. In that case, the formula for obtaining the next approximation breaks down. An example of such a point is $x = 0$ when $f(x) = x^2 - b$. A second situation in which convergence does not occur is depicted in Figure 6.19. If $x_0$ is chosen as the

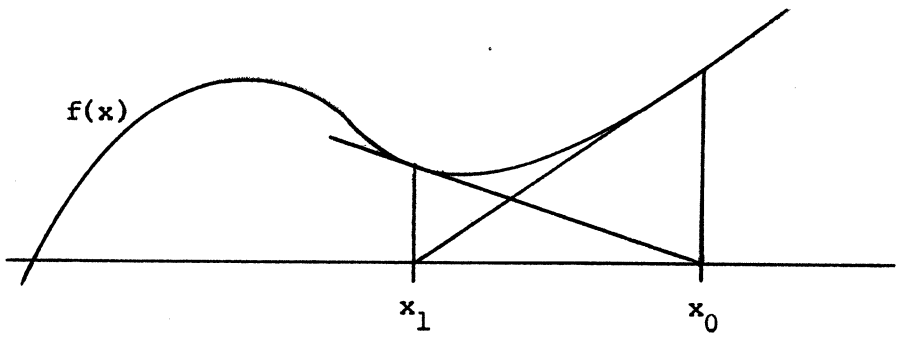



Figure 6.19

starting point, the formula will produce $x_1$ as the next approximation. If that point is now used, $x_0$ is found as the next approximation. Clearly, no progress can be made toward finding a root, since the only two points that can be found are $x_0$ and $x_1$. Such cyclic behavior may occur with three or more points, as well as two.

Since a root may never be found (and, in fact, no real root may exist), one terminating condition for the Newton-Raphson iteration is often the number of iterations performed. Under these conditions, an error message that no root can be found from that starting point (with reasonable work) is usually printed.

Three common conditions used for terminating the iteration are

1) $|f(x)| \leq \varepsilon$

2) $\left|\frac{f(x)}{x}\right| \leq \varepsilon'$ (assuming $x = 0$ is not a root)

3) Change in approximation less (in magnitude) than $\delta$.

All three have the drawback that a false root may be found. However, in contrast with the half interval method there is no terminating condition that guarantees the existence of a zero. The danger of obtaining a false root can be minimized by choosing $\varepsilon$, $\varepsilon'$, or $\delta$ very small.

A flowchart of an algorithm for a Newton-Raphson iteration is shown in Figure 6.20. Data includes the starting point, the function, and the tolerance, $\varepsilon$. The first terminating condition, $|f(x)| \leq \varepsilon$, will be used. The error condition is that no root has been found in 25 iterations. It is assumed that either $f'(x)$ is also given as data or that it can be calculated either analytically or numerically.[6]

A particularly simple implementation of the Newton-Raphson iteration is possible when it is known in advance that the iteration will converge. If either of the first two terminating conditions are used, the iteration can be implemented using a single MAD statement. For example, if $f(x) = x^2 - a$, $x_0 = \frac{a+1}{2}$, and a terminating condition of $\left|\frac{f(x)}{x}\right| \leq \varepsilon$ is chosen, the following complete MAD main routine reads sets of data containing a and $\varepsilon$, and calculates and prints out the square root of a.

```
START       READ AND PRINT DATA          A, EPSLØN
S           THRØUGH S, FØR X = (A+1.)/2., - (X*X - A)/(2.*X),
           1      .ABS.((X*X-A)/X).LE.EPSLØN
            PRINT RESULTS   A, X
            TRANSFER TØ START
            END ØF PRØGRAM
```

Exercises

1. Use the Newton-Raphson Method to calculate, by hand, a root of $f(x) = \sin x + .2$, using $x_0 = 0$ as the starting point and terminating either when $|f(x)| \leq .05$ or after 15 iterations, whichever comes first.

2. Use the Newton-Raphson Method to calculate by hand a root of $f(x) = \sin x$, using $x_0 = 80^\circ\ 57.7'$ as a starting point and terminating when $|f(x)| \leq .05$ or after 25 iterations. (Note $\tan x_0 = 2\pi$)

3. All of the real roots of a polynomial can often be calculated using a Newton-Raphson iteration by first finding one root; then factoring out that root from the polynomial, reducing its degree by one; and then repeating the procedure on the reduced polynomial. Use this procedure to calculate the roots of the following polynomials, using $|f(x)| < .01$ as a terminating condition and any $x_0$ other than an actual root as the starting point.

   a) $x^3 - 5x^2 - x + 5$

   b) $x^3 - 3x - 2$

   c) $x^4 + 10x^3 + 35x^2 + 50x + 24$

4. Using the algorithm presented in this section, write a MAD routine to calculate a root of $f(x) = a \sin x + b \cos x$, where a, b, $\varepsilon$ and $x_0$ are read as data.

---

[6] See Section 6.8.

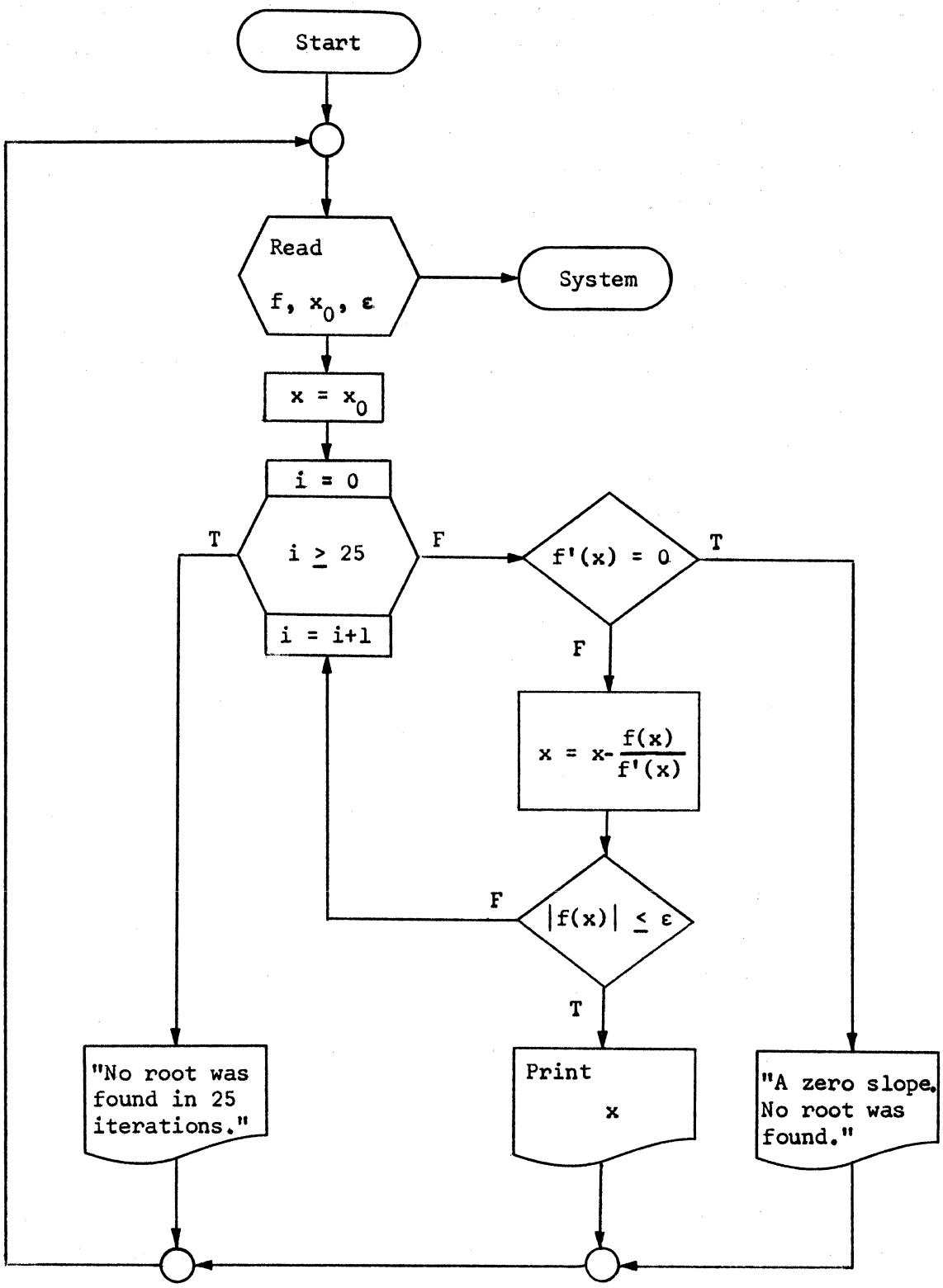

Start
Read
f, $x_0$, ε
System
$x = x_0$
i = 0
i ≥ 25
T
F
i = i+1
f'(x) = 0
T
F
$x = x - \frac{f(x)}{f'(x)}$
F
|f(x)| ≤ ε
T
"No root was found in 25 iterations."
Print
x
"A zero slope. No root was found."

Figure 6.20

5. The Newton-Raphson Method is often used for the calculation of roots of a constant. By choosing the initial guess carefully, an excellent approximation to the root can be obtained in very few iterations.

Draw a flowchart and write a MAD main routine for an algorithm to calculate and printout the real cube root of a real number a. The input data should include a and n ($\geq 0$), the number of Newton-Raphson iterations to be performed.

The procedure that should be used is as follows:

a) Reduce a so as to find b and c such that

$$a = b \cdot 8^c$$

where c is an integer (positive or negative) and $1. \leq b < 8.$

b) Choose a starting point $x_0$ for the Newton-Raphson iteration for the cube root of b. (Note that the cube root of b is between 1. and 2.) Choose as the starting point that point of 10 equally spaced points between 1. and 2. that is closest to the correct answer (e.g. choose $x_0 = 1.05$ for b between 1. and 1.331, i.e. $1.^3$ and $1.1^3$).

c) Perform n Newton-Raphson iterations.

d) Obtain the answer by multiplying the results of part c by $2^c$ and introducing the correct sign.

Note that n = 4 will give answers correct to at least 6 decimal places.

## 6.6 Nested Loops

Many algorithms are best designed with nested loops, i.e. loops within the scope of other loops. Nested loops often involve repeating iterative calculations for a set of values of at least one variable. For example, an algorithm for calculating the square root of all integer values from 1. to 100. could easily be obtained using a Newton-Raphson iteration to calculate the square root and another loop to vary the number, the root of which is to be calculated. The flowchart and MAD routine of Figure 6.21 illustrates such a calculation, making use of the algorithm of Section 6.5. The second loop, i.e. the loop for the Newton-Raphson iteration, is referred to as the inner loop since it is entirely a part of the scope of the outer loop. The inner loop is traversed as many times as is necessary to satisfy the test (for that loop) each time the outer loop is traversed. The absolute value operation in the terminating condition in Section 6.5 was omitted here, since x is always greater than the actual square root, and thus $x^2$-a is thus always positive.

In this example, an iteration statement appears within the scope of another iteration statement. There are two limitations on the nesting of loops. First, any loop that occurs within the scope of another loop must be entirely within the scope of the outer loop. More than one scope may terminate on the same statement, although this arrangement should be avoided. Figure 6.22a illustrates two nested loops. Each of the statement sets in both Figure 6.22a and b may include other complete iteration statements or complete compound conditional statements. Although the form in Figure 6.22b is legal, it should usually be avoided by using the first form with a CØNTINUE statement as <statement-set>$_3$.

If the scopes of more than one loop end at the same statement, then the variable in the last THRØUGH statement whose scope ends at that statement is incremented and the test in that THRØUGH statement is evaluated. If its value is false, then the scope of that statement is executed. Otherwise (when the test is satisfied, i.e. the expression is true), the variable in the preceding THRØUGH statement, the scope of which ends at the common statement, is incremented and the test in that statement is made. Only when the tests in all THRØUGH statements having the common scope termination are true is the statement following the scope termination executed.

The second limitation on the use of iteration statements is that no iteration statement may be in the scope of more than 50 other iteration statements. (This restriction is rarely a problem.)

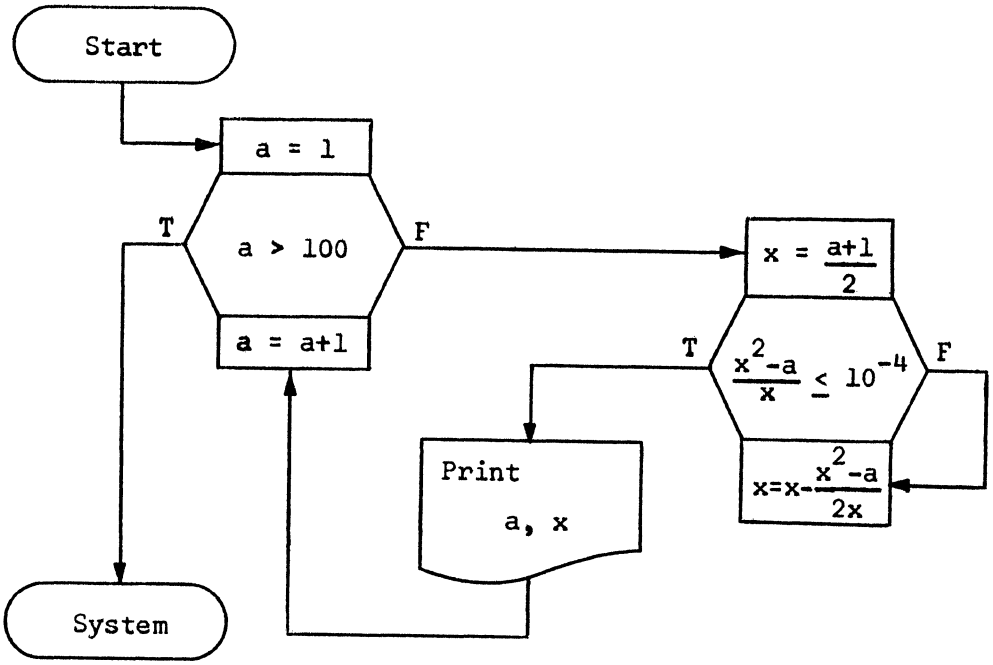



Figure 6.21a

```
          RRØUTINE TØ PRØDUCE A TABLE ØF SQUARE RØØTS
           THRØUGH S1, FØR A = 1.,1.,A.G.100.
S2              THRØUGH S2, FØR X = .5*(A + 1.),-(X*X-A)/(2.*X),
          1         (X*X-A)/X .LE. 1E-4
                  PRINT RESULTS A,X
S1                CØNTINUE
           END ØF PRØGRAM
```

Figure 6.21b

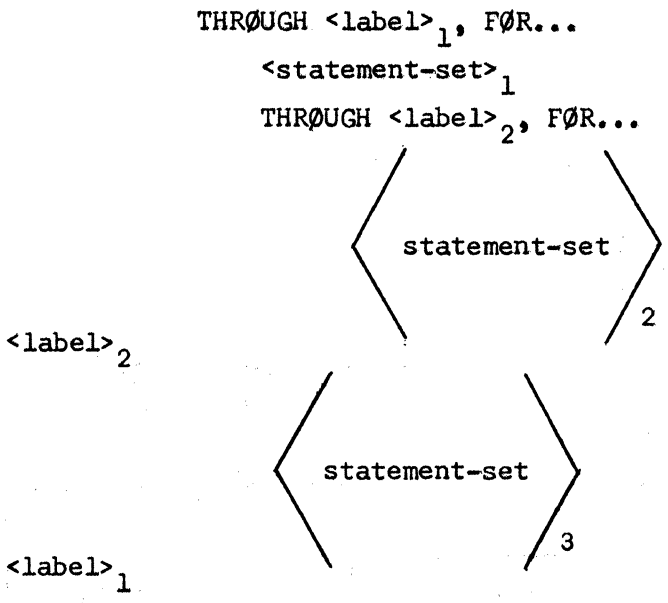



Figure 6.22a

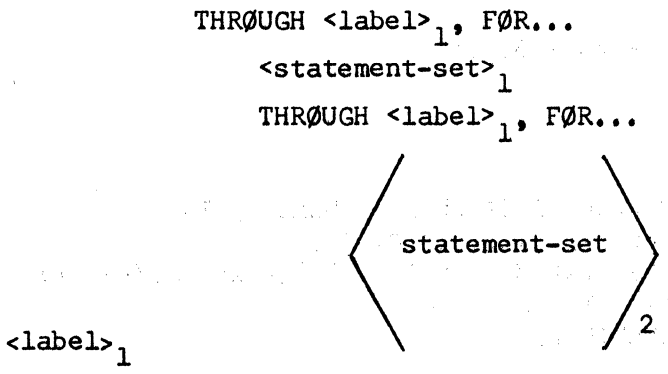



Figure 6.22b

The algorithm of Figure 6.21 can be extended to calculate and print a table of the second through eighth roots of the integer values from 1 to 100. A flowchart and the corresponding MAD routine are shown in Figure 6.23. Each line of the table printed will be preceded by a line

X(1)...X(8)

This cannot be avoided without the use of formatting or at least the storing of all the results in an array.

Many algorithms require nested loops as part of the basic calculations. An example of such a problem is the following:

> Find and print out all permutations of the three integers 1, 2 and 3. There are six possible permutations, namely 123, 132, 213, 231, 312, 321. We will let a general such permutation be represented by ijk and it is required that $i \neq j$, $j \neq k$, and $k \neq i$. Although the solution of this problem is known, the algorithm developed is typical of many more complex algorithms.

One algorithm that could be developed would involve three nested loops, one for each of the variables. Each variable is stepped through the three values, obtaining each of the possible sets of three numbers 1, 2, and 3. Each set is then tested to see whether all three integers are different. Only those sets passing this test are printed. A flowchart for this algorithm is shown in Figure 6.24.

This is a wasteful algorithm since many cases are considered that could not be accepted. For example, the first time the inner loop is entered, i and j are equal to 1 and so the test will be false for all values of k. The algorithm can thus be modified to split the test and omit the third loop in cases when i equals j. Furthermore, the inner loop may be replaced by a simple calculation (k = 6-i-j), since the sum of i, j and k must be 6 for all sets of values satisfying the conditions. Thus, an improved algorithm, the flowchart and MAD main routine for which are shown in Figure 6.25, can be obtained.

Finally, it should be noted that algorithms discussed in this section have a second thing in common in addition to the use of nested loops. None of these algorithms require any input data; they are designed to solve a specific problem, rather than a class of problems.

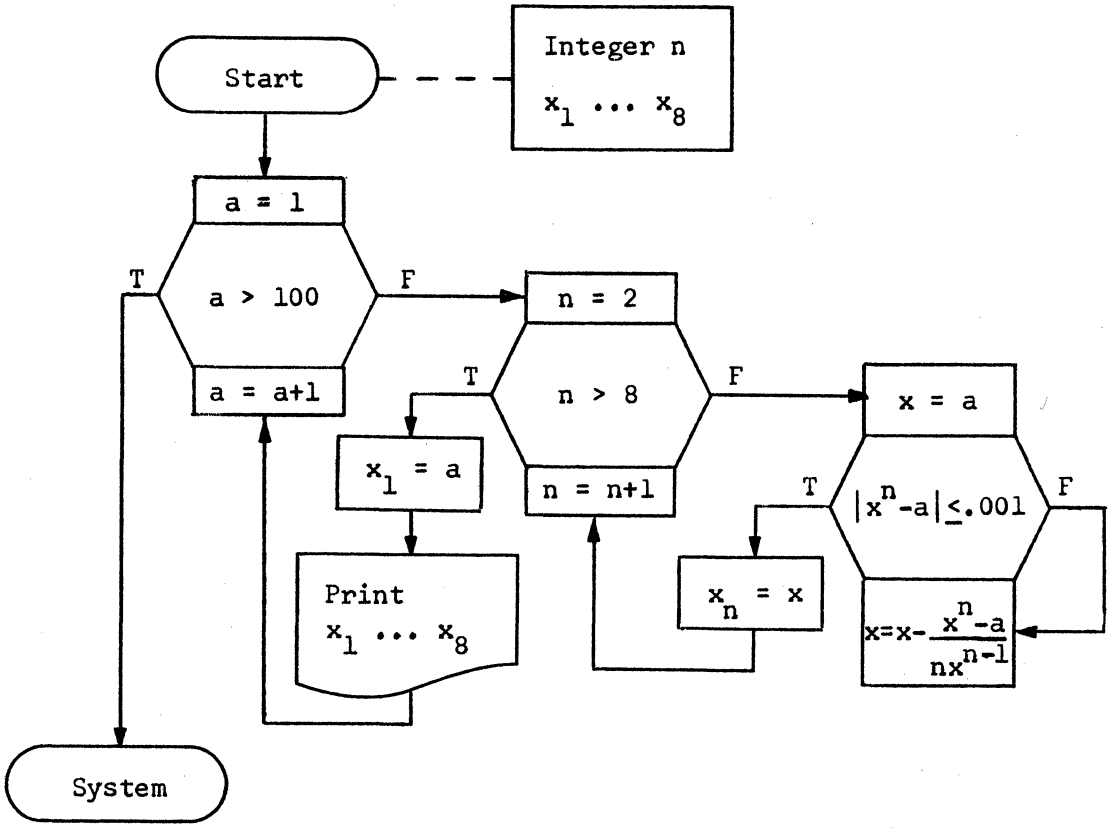



Figure 6.23a

```
      RRØUTINE TØ CALCULATE AND PRINT SECØND THRØUGH
      R    EIGHTH RØØTS ØF FIRST HUNDRED INTEGRAL VALUES
      RX(I) IS THE I-TH RØØT ØF X(1)
       INTEGER N
       DIMENSIØN X(8)
       THRØUGH S1, FØR A = 1.,1.,A.G.100.
           THRØUGH S2, FØR N = 2,1,N.G.8
S3                THRØUGH S3, FØR X = A,-(X-A/X.P.(N-1))/N,
1                     .ABS.(X.P.N-A).LE.1E-4
                  X(N) = X
S2                CØNTINUE
                  X(1) = A
                  PRINT RESULTS X(1)...X(8)
S1              CØNTINUE
                END ØF PRØGRAM
```

Figure 6.23b

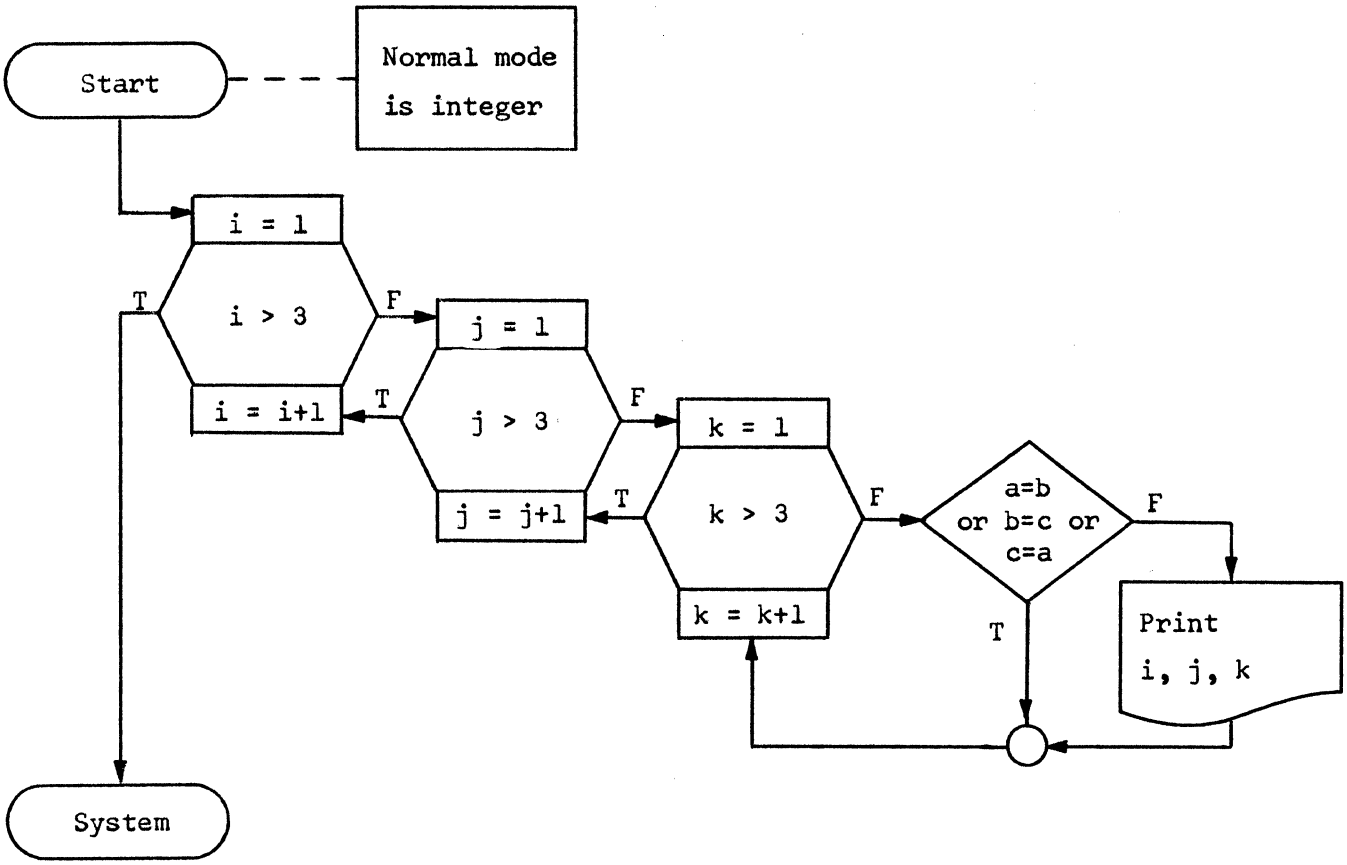

Start
Normal mode
is integer
i = 1
i > 3
T
F
i = i+1
T
j = 1
j > 3
F
j = j+1
T
k = 1
k > 3
F
k = k+1
T
a=b
or b=c or
c=a
F
T
Print
i, j, k
System

Figure 6.24

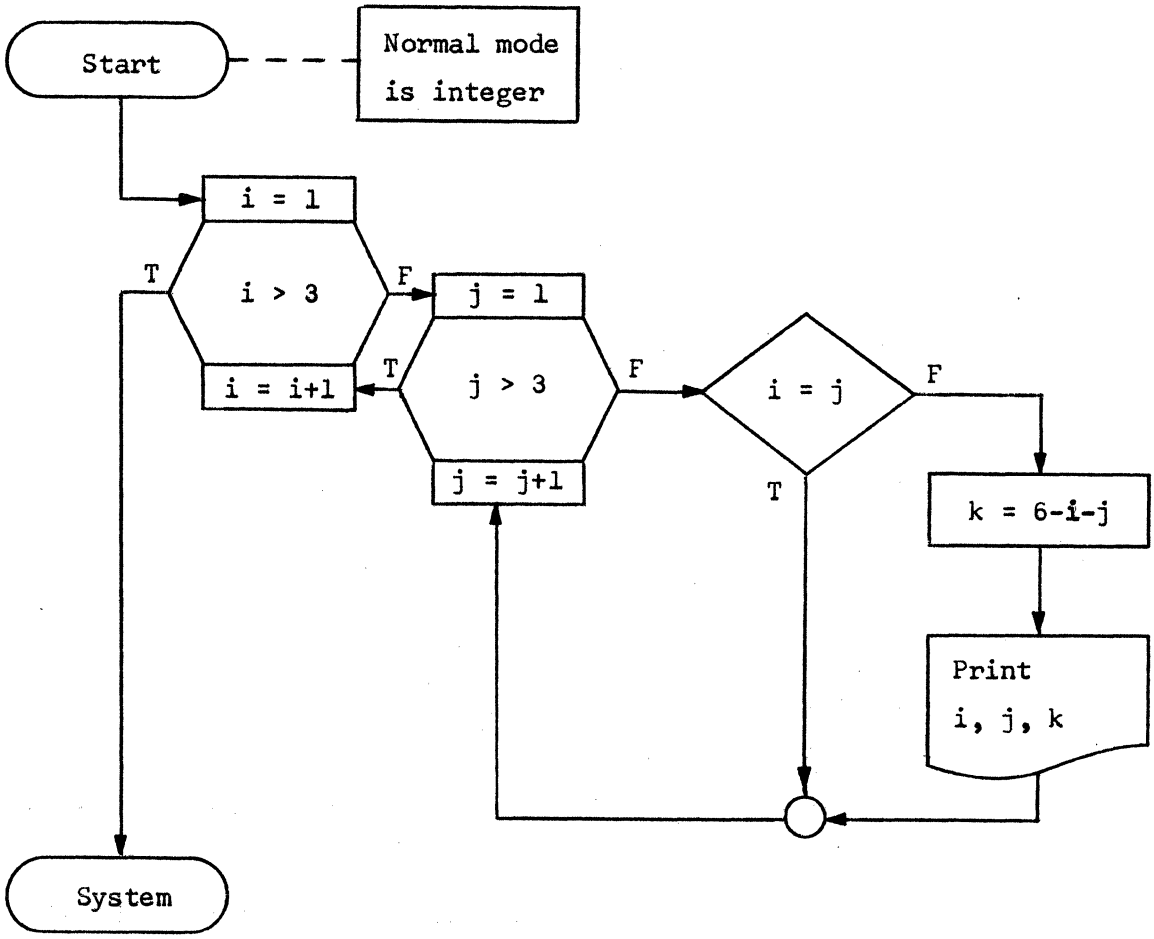



Figure 6.25a

```
      RRØUTINE TØ FIND THE PERMUTATIØNS ØF 1, 2, 3
       NØRMAL MØDE IS INTEGER
       THRØUGH S1, FØR I = 1,1,I.G.3
           THRØUGH S2, FØR J = 1,1,J.G.3
               WHENEVER I.E.J, TRANSFER TØ S2
               K = 6-I-J
               PRINT RESULTS I,J,K
S2             CØNTINUE
S1         CØNTINUE
           END ØF PRØGRAM
```

Figure 6.25b

Exercises

1. In the routine of Figure 6.25b, how many times is each of the Boolean expressions evaluated?

2. Draw a flowchart and write a MAD main routine to calculate and print the interest and total on deposit in a savings account each quarter. Interest is 4%, compounded quarterly. Print also on each line the number of years the account has been on deposit. The account was started at $1000. and printing is to be terminated when the deposit has doubled.

3. Modify Exercise 2, so as to print tables for interest rates of 2%, 3%, 3 3/4%, 4%, 4 1/4% and 4 1/2%.

4. Modify the algorithm presented in this section to find all the permutations of the letters A, B and C. Write the MAD routine, using PRINT BCD RESULTS for the output statement.

5. Examine the following program.

```
$CØMPILE MAD, EXECUTE
                INTEGER N
                READ AND PRINT DATA
                N = 0
                THRØUGH FINIS, FØR X = 1., X, X .G. .ABS.Z
                     Y = X
FINIS                N = N + 1
                PRINT RESULTS N, Y, Z
                END ØF PRØGRAM
```

As far as we know, it is a correct program to do something with an input datum Z. What does it do? What are outputs N and Y? How are they related to Z?

6. Write an iteration statement terminating in a statement labeled FINIS that will take index X, initialize it at value Y, and multiply it by a constant, K, each time through the loop, until the product K*X is greater than a given Z.

7. The following is a legal MAD program, with an appropriate set of data.

```
                DIMENSIØN A(20), B(20)
                INTEGER A, B, I, J, N
                J = 0
                READ AND PRINT DATA
                THRØUGH SAD, FØR I = 0, 1, I .G. N
                    WHENEVER A(I) .E. A(I)/2*2
                         B(J) = A(I)
                         J = J + 1
SAD                 END ØF CØNDITIØNAL
                PRINT RESULTS B(0) . . . B(J-1)
                END ØF PRØGRAM
$DATA
N = 6, A = 3, -4, 6, 6, -7, 12, -16*
```

a) What does this program do?

b) Draw a detailed flowchart of this program.

c) Rewrite the program to perform exactly the same job, but do not use any THRØUGH statements.

8. For the following program

a) What data is needed?

b) What DIMENSIØN statement is needed?

c) What does the program do?

d) How many times are the instructions labeled S and T executed?

```
          READ AND PRINT DATA
          INTEGER I, J
          SUM = 0.0
          THRØUGH S, FØR I = 0, 10, I .G. 90
              SUB(I/10) = 0.0
              THRØUGH T, FØR J = I, 1, J .G. I+9
                  SUB(I/10) = SUB(I/10) + A(I)
T                 CØNTINUE
              SUM = SUM + SUB(I/10)
S             CØNTINUE
          PRINT RESULTS SUB(0) ... SUB(9), SUM
          END ØF PRØGRAM
```

9. For the following program, what is printed out as a result of the print instruction? What does it do?

```
          THRØUGH DØNE, FØR VALUES ØF GAIN = 10., 25., 50.
S         BAL = 1000.00
          FINAL = 1000.00*(1+GAIN/100.)
          THRØUGH END, FØR YRS = 0, .25, BAL .GE. FINAL
              BAL = BAL + BAL*0.10
END           CØNTINUE
DØNE      PRINT RESULTS GAIN, YRS, BAL
          END ØF PRØGRAM
```

How many times are the statements S and END performed?

10. Each of the following programs was written by a programmer who did not quite understand the logic of the problem or the MAD language, and in addition was quite careless. Consequently, a number of mistakes may be found in the programs. Indicate all the errors you can find in both the syntax of the routine and the data. Then, list separately those errors which are of a logical nature and would not be caught by the MAD compiler. Finally, list any bad usages.

a) The following MAD program was supposed to input and examine two vectors V and W, each having n + 1 components with subscripts 0, 1, ..., N and copy the larger - in absolute value - of each corresponding component into a new vector X. For example, if V(5) = 6 and W(5) = -10, then the program should set X(5) = -10. It then was to output the vector X. N, V, and W are the input data.

```
        READ & PRINT DATA
        N = 100
        DIMENSIØN V(N), W(N)
        INTEGER I, N
        THRØUGH BLAST, FØR I = 0, 1, I .GE. N
            X(I) = V(I)
BLAST       WHENEVER .ABS. W(I) .G.ABS.V(I), W(I) = X(I)
        PRINT RESULTS X0,...,XN
        END ØF PRØGRAM
$ DATA N = 3 V = 0.0, 7.2, -1.4, 3.1, W = 1, 0, 2.6, 1.3, -4.6
```

b) This program is designed to find the largest of 10 numbers A(1) ... A(10) as well as the sum of the squares of these numbers.

```
        READ AND PRINT
        DIMENSIØN A(9)
        LGST = 0
        SUMSQRS = 0
        THRØUGH S FØR I = 1, 1, 1.L.10
            WHENEVER A(I) .G. LGST, LGST = A(I)
            SUMSQRS = SUMSQRS + A(I)A(I)
S           END ØF CØNDITIØNAL
        PRINT RESULTS LRGST, SUMSQRS
        TRANSFER TØ START
        INTEGER I
        END ØF PRØGRAM
$DATA
        A = 1., 2., 3., 4., 5., 6., 7., 8., 9., 10.*
```

## 6.7 Approximating Areas under Curves

The problem discussed in this section involves finding the area under a curve defined by a function f(x) which is never negative in the range $a \leq x \leq b$. The area is bounded below by the x-axis, on the left by a vertical line at x = a and on the right by a vertical line x = b. Such an area is shown in Figure 6.26a. The mathematical treatment of this problem is included in the calculus in connection with integration theory. Using the techniques developed there, such areas can often be found by exact analytic methods. In general, however, these areas cannot be found analytically and so approximations to them must be used. Elementary methods for computing such approximations are presented in this section. Most of the treatment can be understood without the calculus, but it will, of course, have more meaning to those who have studied that subject.

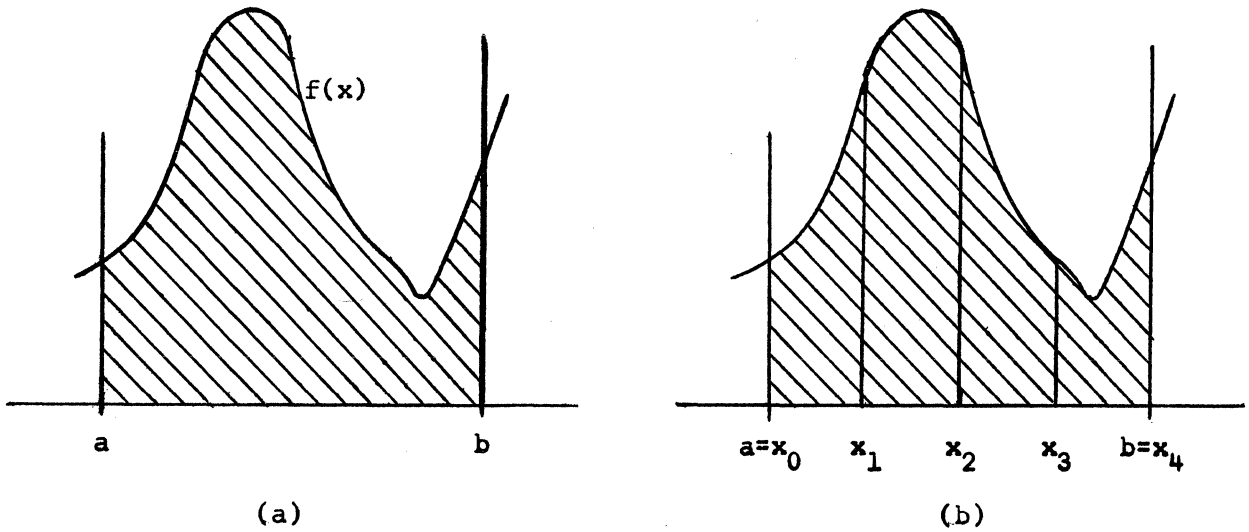



Figure 6.26

The methods to be considered here will all involve partitioning the interval from a to b into several subintervals of equal length. We will assume that n such subintervals are used and let $x_0 = a$ and $x_n = b$. The endpoints of these subintervals are then denoted by $x_0, x_1, \ldots, x_{n-1}, x_n$ where $x_{i+1} - x_i$ is a constant, say h, for all i. We then have $h = (b-a)/n$, and the i-th subinterval is bounded by $x_{i-1}$ and $x_i$. This type of partition into subintervals is illustrated in Figure 6.26b. It is now assumed that the value of the function is known, or can be computed as needed, at each of the points $x_i$. The methods vary only in how this information is used to find an approximation to the area. In each case the function is approximated in one or two of the subintervals by a curve (e.g. a straight line), under which

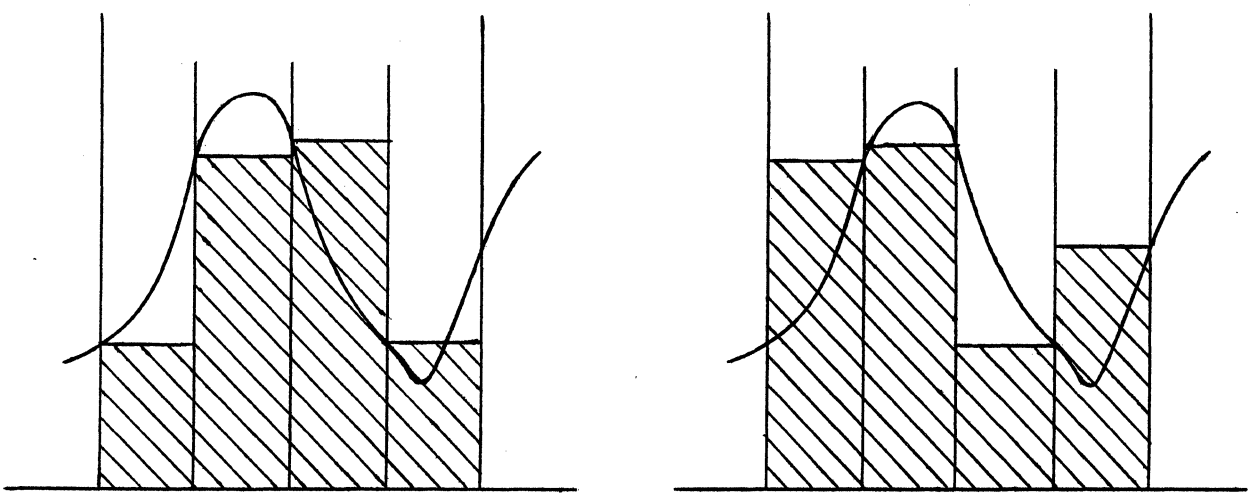
the area is known. In the first two methods the function is approximated by its value at one end of the interval, and thus the area in the subinterval is just the area of a rectangle. The first method produces what is called a left sum by using the approximation $\hat{f}(x) = f(x_{i-1})$ in the interval $x_{i-1} < x < x_i$ and thus the rectangle is of width h and height $f(x_{i-1})$. The left sum is then given by

$$[f(x_0) + f(x_1) + \ldots + f(x_{n-1})] \cdot h$$

The second method produces the right sum by using $\hat{f}(x) = f(x_i)$ in the interval $x_{i-1} < x < x_i$ and gives the approximate area

$$[f(x_1) + \ldots + f(x_{n-1}) + f(x_n)] \cdot h$$

The rectangles used in these two cases for our example function with four subintervals are shown in Figure 6.27. Intuitively it seems clear that the more subdivisions which are used the closer the approximation obtained will be to the actual area and this can indeed be proved mathematically. Note that these two areas differ only in the choice of the end point which is involved in the sum.



Left Sum (a)

Right Sum (b)

Figure 6.27

A somewhat better approximation can be obtained by using trapezoids to compute the subareas. In each case the function is then approximated by a straight line connecting the values at the endpoints of each subinterval. Thus each approximate subarea is given by

$$[f(x_{i-1}) + f(x_i)] \cdot \frac{h}{2}$$

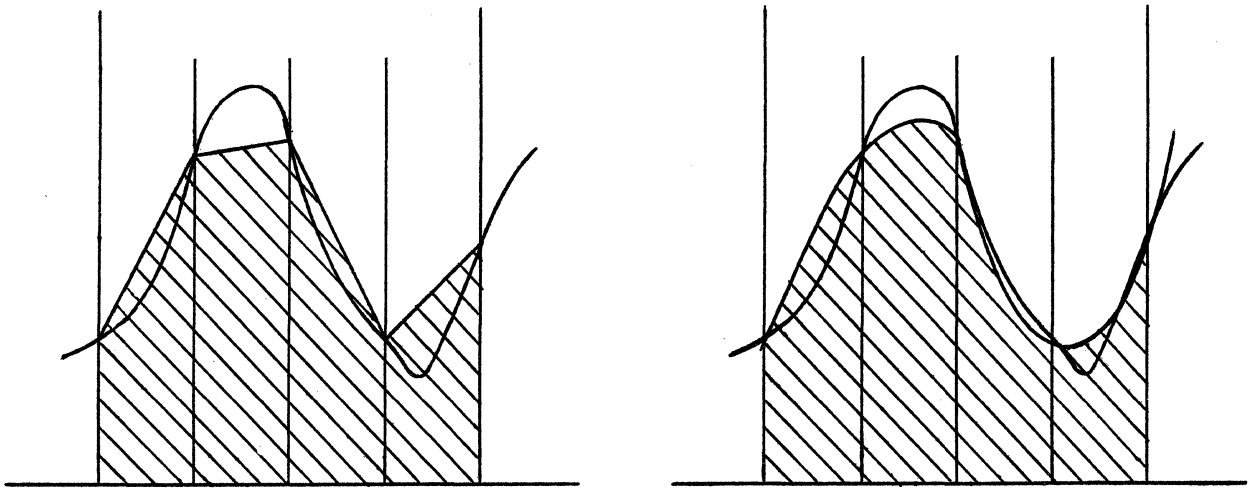
which are the areas of the trapezoids as illustrated in Figure 6.28a. On summing these areas we obtain the formulas

$$[f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n)] \cdot \frac{h}{2}$$

or

$$\left[\frac{f(x_0)}{2} + f(x_1) + \ldots + f(x_{n-1}) + \frac{f(x_n)}{2}\right] \cdot h$$

Note that this sum is identical with the left sum and right sum except that the average of the values of the function at endpoints is used. This method is usually called the trapezoidal rule for numerical integration.



Trapezoidal Rule
(a)

Simpson's Rule
(b)

Figure 6.28

The final method presented here uses a parabola to approximate the curve over two adjacent subintervals. It can be shown easily by the calculus that the area under this parabola (which passes through the points $(x_{i-1},f(x_{i-1}))$, $(x_i,f(x_i))$ and $(x_{i+1},f(x_{i+1}))$ is given by

$$\left[f(x_{i-1}) + 4f(x_i) + f(x_{i+1})\right] \cdot \frac{h}{3}$$

Thus the total area under the curve is given by

$$[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)] \cdot \frac{h}{3}$$

This method of numerical integration is commonly called Simpson's rule. Figure 6.28b shows the areas which are used to make the approximation in our example curve. Note

that in this case the number of subintervals must be even or one of them must be treated separately by some other method. In Figure 6.29 we show the area under the curve of $x + \frac{4}{x^2} - 2$ between x = 1 and x = 4 and a flowchart for computing this area using both the trapezoidal rule and Simpson's rule. Since this area can easily be shown to be 4.5 by the calculus, finding it is not a productive computational problem. It is a useful computation, however, to study the behavior of integration done approximately on the computer. In this example, we use nested loops to compute the integral for 2, 4, 8, ..., 65536 and 131072 subintervals. Clearly two intervals are not enough, but how many should be used to achieve a given accuracy is not at all clear. Because of the finiteness of the number representation normally used in machines one can achieve only limited accuracy by these approximations, although multiple precision may be used to get greater accuracy. The use of more subintervals than are necessary to produce this limited accuracy is not only a waste of computer time, but in fact will produce less accurate results. Thus the actual convergence of these computational sums to the area does not behave like the basic mathematical theory might suggest. Mathematical theory concerning the roundoff errors which cause this difference does exist and is one of the topics treated in numerical analysis.

An actual MAD program and the results produced for the example mentioned above are shown in Figure 6.30. The comparison of M and M/2*2 determines whether M is odd or even. There are more efficient ways of writing the algorithm for Simpson's rule and one of these will be suggested in the exercises. In studying the results one notes that the best approximation for both integration rules occurs with 16384 subintervals. Simpson's rule consistently gives a better answer in this example, although it is possible to construct functions for which this not true. In either case the answers get progressively worse as more intervals than the optimal number are used. Theoretically it is easily seen that the trapezoidal rule should always produce results larger than the actual area in this example; however, the results in this example actually drop below 4.5 if an excessive number of subintervals are used.

When defining a definite integral in the calculus, subintervals such as we have used in this section are employed, but the subintervals introduced there are not required to be of equal width. The lower sum of the calculus is obtained by choosing the smallest of the values of f(x) which occurs in a subinterval as the approximation of the function in that subinterval. The upper sum of the calculus is similarly obtained using the largest value of f(x) which occurs in the subinterval. The upper and lower sums for our example curve using four subintervals are shown in

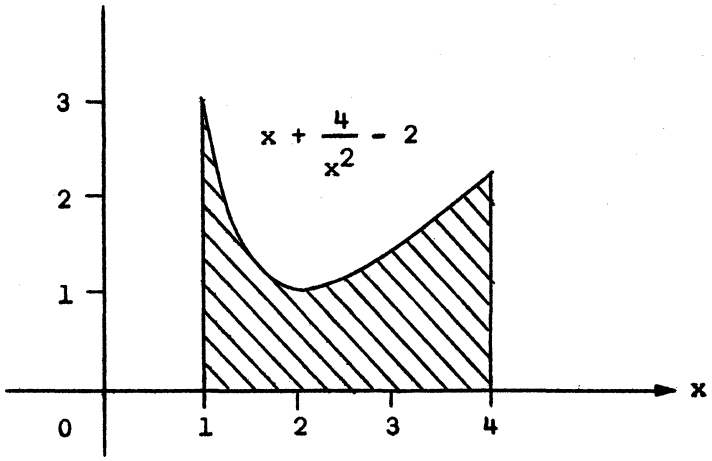



Figure 6.29a

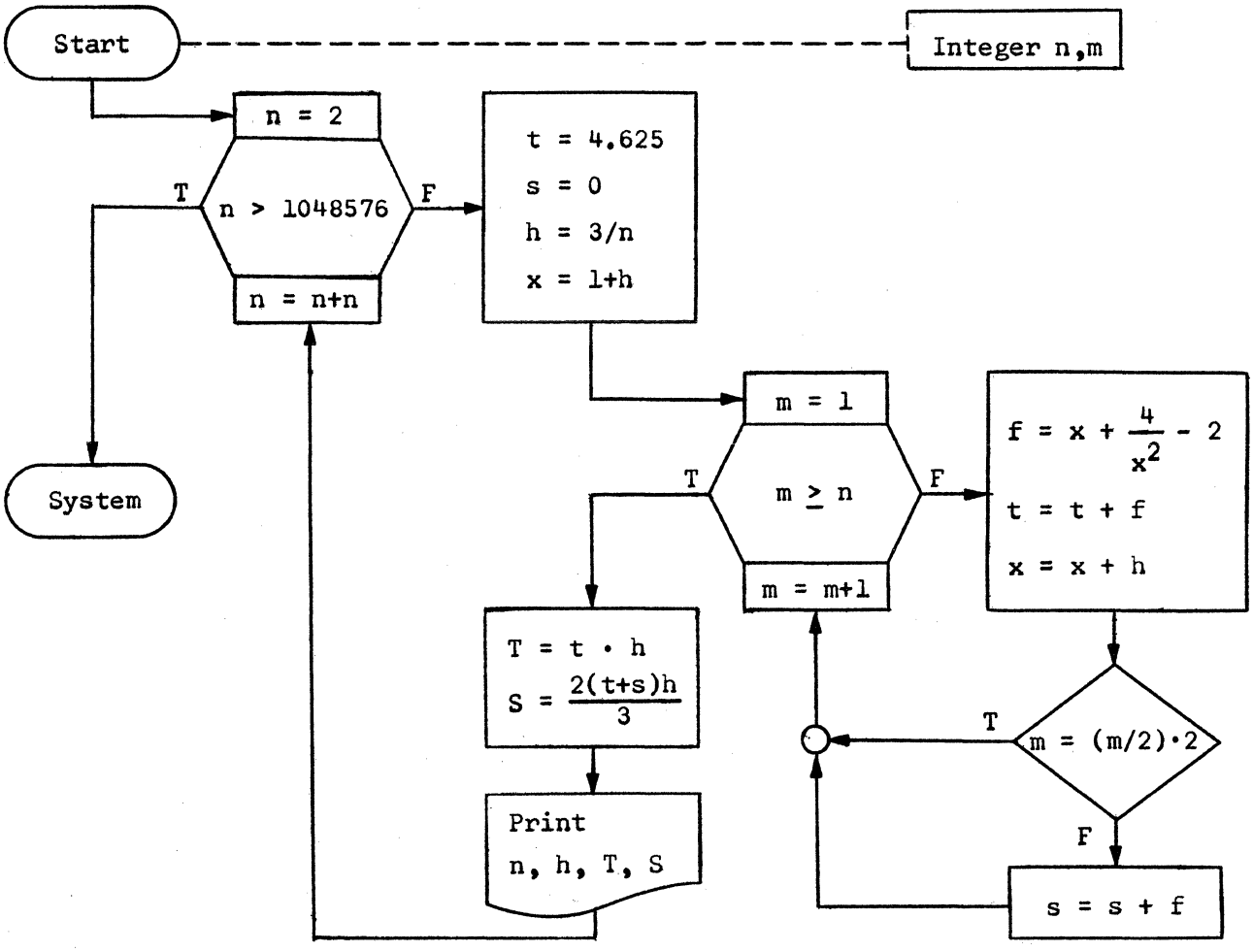


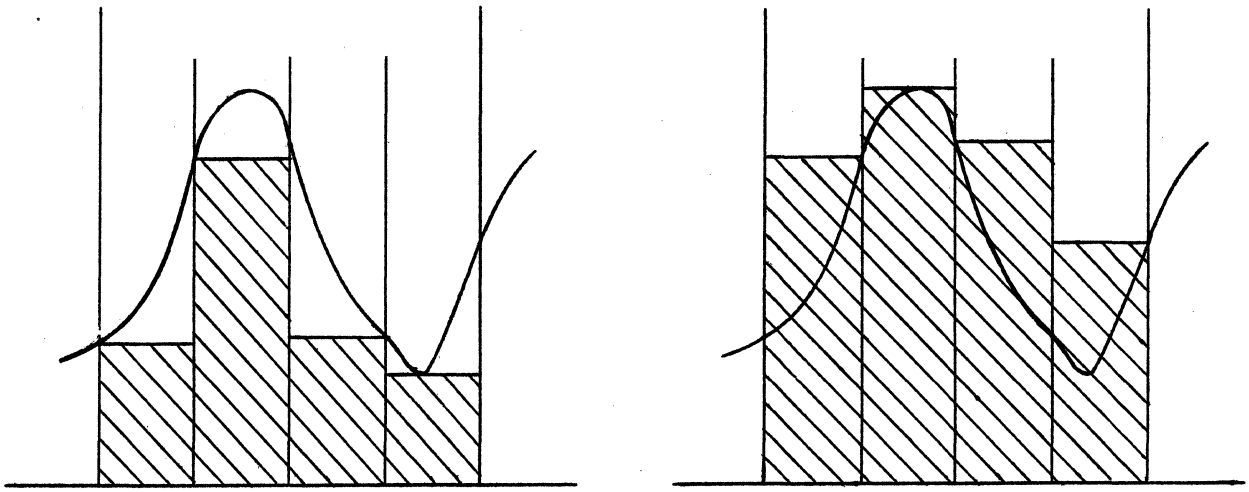

Figure 6.29b

```
      PROGRAM TO FIND AREA UNDER X+4/(X.P.2)-2 BETWEEN X=1 AND X=4
       INTEGER N, M                                                          *001
       PRINT COMMENT $1$                                                     *002
       THROUGH FIND, FOR N = 2, N, N .G. 1048576                             *003
           T = 4.625                                                         *004                01
           S = 0.0                                                           *005                01
           H = 3.0/N                                                         *006                01
           X = 1.0 + H                                                       *007                01
           THROUGH SUM, FOR M = 1, 1, M .GE. N                               *008                01
                F = X + 4.0/(X*X) - 2                                        *009                02
                T = T + F                                                    *010                02
                WHENEVER M .NE. M/2*2, S = S + F                             *011                02
                X = X + H                                                    *012                02
      SUM       CONTINUE                                                    *013                02
       TAREA = T * H                                                         *014                01
       SAREA = 2.0*(T+S)*H/3.0                                               *015                01
FIND   PRINT RESULTS  N, H, TAREA, SAREA                                     *016                01
       END OF PROGRAM                                                        *017

N =          2,   H =      1.500000,   TAREA =      8.647500,   SAREA =      6.905000
N =          4,   H =       .750000,   TAREA =      6.337365,   SAREA =      5.567320
N =          8,   H =       .375000,   TAREA =      5.339875,   SAREA =      5.007379
N =         16,   H =       .187500,   TAREA =      4.897910,   SAREA =      4.750589
N =         32,   H =       .093750,   TAREA =      4.693257,   SAREA =      4.625039
N =         64,   H =       .046875,   TAREA =      4.595190,   SAREA =      4.562502
N =        128,   H =       .023438,   TAREA =      4.547234,   SAREA =      4.531249
N =        256,   H =       .011719,   TAREA =      4.523525,   SAREA =      4.515622
N =        512,   H =  5.859375E-03,   TAREA =      4.511735,   SAREA =      4.507808
N =       1024,   H =  2.929688E-03,   TAREA =      4.505853,   SAREA =      4.503896
N =       2048,   H =  1.464844E-03,   TAREA =      4.502907,   SAREA =      4.501933
N =       4096,   H =  7.324219E-04,   TAREA =      4.501417,   SAREA =      4.500937
N =       8192,   H =  3.662109E-04,   TAREA =      4.500636,   SAREA =      4.500408
N =      16384,   H =  1.831055E-04,   TAREA =      4.500174,   SAREA =      4.500084
N =      32768,   H =  9.155273E-05,   TAREA =      4.499797,   SAREA =      4.499800
N =      65536,   H =  4.577637E-05,   TAREA =      4.499322,   SAREA =      4.499419
N =     131072,   H =  2.288818E-05,   TAREA =      4.498509,   SAREA =      4.498749
N =     262144,   H =  1.144409E-05,   TAREA =      4.496956,   SAREA =      4.497458
N =     524288,   H =  5.722046E-06,   TAREA =      4.493896,   SAREA =      4.494908
N =    1048576,   H =  2.861023E-06,   TAREA =      4.487854,   SAREA =      4.489864
```

Figure 6.30

Figure 6.31. It is shown in the calculus that the upper and lower sums approach the same value as the maximum width of the subintervals becomes arbitrarily small. This value is called the definite integral of the function over the interval and equals



Lower Sum
(a)

Upper Sum
(b)

Figure 6.31

the area under those functions which are non-negative. Since finding the maximum or minimum value which a function f(x) takes on in a given interval would require a great deal of work on the computer, the upper and lower sums would be very difficult to compute. It is easily seen that all four of the approximations we have used are always between the upper and the lower sums for the same set of subintervals. Thus, theoretically these values will also approach the definite integral and all of them are easily computed. In the definition of the definite integral there is no restriction on f(x) being non-negative and so these same techniques are valid for approximating definite integrals in general. Our restriction to functions which were never negative was made only to simplify the discussion of areas.

There are numerous other more refined methods of computing approximate areas under curves and other integrals. These methods along with study of their accuracy and efficiency have been studied in depth and considerable material is available on them in various books[7] on numerical analysis.

[7] Hildebrand, F. B. Introduction to Numerical Analysis, McGraw-Hill, New York, 1956 Chapter 8.

Exercises

1. Calculate by hand, the area under $f(x) = \sin x$ in the interval $0 \leq x \leq \pi$, by each of the four algorithms discussed in this section for

a) 2 intervals

b) 4 intervals

c) 6 intervals

d) 12 intervals

2. Draw a flowchart and write a MAD program to perform the calculations of problem 1 continuing on to 20, 90, and 180 subintervals. It would be instructive to run that program on a computer using different approximations for $\pi$ (e.g. 3.14, 3.1416, and 3.1415927.

3. Rewrite the algorithm for the trapezoidal rule and Simpson's rule which was given in the text to take two intervals at a time. This will cut down the time used in loop control and deciding whether M is even or odd. Thus, a more efficient algorithm can be produced.

## 6.8 Approximating Slopes of Curves

The slope of a curve is a local measure of the vertical change as compared with the horizontal change. The slope (or derivative as it is called in the calculus) of a function f at a given value x is defined as the limit of the ratio

$$\frac{f(x+h) - f(x)}{h}$$

as h approaches zero, if this limit exists. Geometrically this ratio is illustrated in Figure 6.32. The h may be positive or negative and the same limit must be approached regardless of how h approaches zero. The slope of the function f(x) at x is equal to the slope of a tangent to f(x) at x if a unique tangent can be drawn. This tangent is also shown in Figure 6.32.

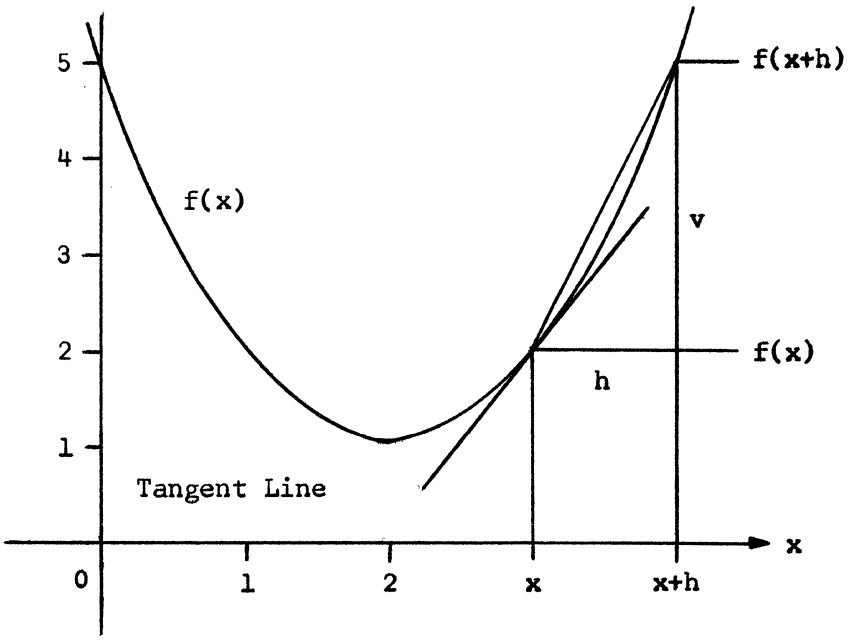



Figure 6.32

In this section we will show some of the problems which develop if we try to find the limit of this ratio on a computer by letting h get smaller and smaller. The example we use involves the polynomial (quadratic to be specific) function $e(x) = x^2 - 4x + 5$ which is very simple and well behaved mathematically. Since this function is also very easy to compute, any problems which occur cannot be blamed on the complexity of the problem chosen.

Before proceeding with the example it is worth pointing out that the derivative of the general polynomial

$$a(x) = a_0 + a_1x + \ldots + a_nx^n = \sum_{i=0}^{n} a_i x^i$$

is shown in the calculus to be given by

$$a'(x) = a_1 + 2a_2x + \ldots + na_nx^{n-1} = \sum_{i=0}^{n-1} (i+1)a_{i+1}x^i$$

which is again a polynomial. Applying this to our sample function we get

$$e'(x) = 2x - 4$$

which means that we have a very easy way of computing what our answer should be. The algorithm we present is a general one, designed for arbitrary polynomials, and our particular quadratic is introduced as data. We will experiment only at the values x = 2.0, x = 3.0 and x = 4.0 where both the value of the function and the slope are exactly known and exactly computed by the machine. Furthermore, we will let h = 1.0 initially and cut it in half at each step of the iteration. Thus, x + h will be exactly represented in the computer for the first 25 or 26 iterations. On the 27-th iteration (26-th for x=4.0), x + h equals x, since $2 + 2^{-26}$ requires 28 significant bits. In all but the last few iterations, the x + h, f(x) and h going into the evaluation of the ratio

$$\frac{f(x+h) - f(x)}{h}$$

are all exact. Since h is an exact power of two, division by h will only alter the characteristic of the number obtained. The MAD program for this algorithm is shown in Figure 6.33a. Following the computation and printing of the value of the polynomial POL and its derivative DER, the successive values of H and the corresponding values of the approximation to the slope are printed. The process is terminated when we get a result close enough (within a small value EPS provided as data) to the derivative or after 29 iterations (i.e. M=28). This latter condition is included in case the process does not produce a result which is within EPS of the correct answer. Part of the first set of data giving the degree N and coefficients A of our sample polynomial as well as EPS = .0001 is included with the program listing. The value of x used is shown with each set of results. Now to look at the results. The results turn out quite well when X = 4.0 as shown in Figure 6.33b. An approximation of 4.000000 accurate to 6 decimal places is obtained with M = 13. In the second set of data we need supply only X = 3.0 and repeat the process. This time the result should be 2.0, but some trouble develops as shown in Figure 6.33c. When M = 12 and

and 13 we get an approximation of 2.000244 which is almost within our allowed error; however, the process then begins to diverge from the answer. Note that because of the peculiarities of the computer's arithmetic the same values are retraced as it diverges that were produced when it was getting closer. When M = 26, H is completely overshadowed by X in X + H and the result is zero. Thus the process never gets within .0001 of the correct answer. Finally, we let X = 2.0 where the derivative is 0 and observe a somewhat different result as shown in Figure 6.33d. Here once again it converges toward the correct result getting almost within the tolerance we have allowed, but then diverges. When we get a zero result caused by the loss of accuracy in X + H, .ABS.(SLOPE-DER) is satisfied with M = 26 and the process is stopped. Thus, the correct answer is produced, but for the wrong reason.

All of the problems which have developed in this example are caused by the fact that the computer uses only a limited and fixed representation for numbers. This means that only limited accuracy is maintained and the bits beyond a certain position must be thrown away or used to round the number by some scheme. Actually in the IBM 7094 on which this problem was run, the bits are simply thrown away and so the number is truncated with just 27 bits left in the mantissa. Essentially the same characteristics hold on a decimal machine.

This example has been included to demonstrate to the reader that things are not always what they might seem to be when done on a computer. Extreme caution must always be maintained to make sure that the numbers being produced are significant. Careful analysis of the problem, the process used to solve it, the magnitude of the numbers involved at the various stages in the computation and many other factors must be made.

## Exercises

1. Draw a flowchart for the algorithm presented in this section.

2. A better approximation to the slope of a curve at a given point can usually be obtained by using an interval which has that point as its midpoint. Rewrite the algorithm of this section to compute the value of the polynomial at x - h as well as at x + h where h takes on positive values as before. Using values of the function at the three points, compute a left approximation to the slope given by (f(x) - f(x-h))/h, a central approximation given by (f(x+h) - f(x-h))/2h and the right approximation used in the given algorithm. Compare the results.

```
            PROGRAM TO FIND THE APPROXIMATE SLOPE
            OF A LINE TANGENT TO A POLYNOMIAL.

      INTEGER I, M, N
      DIMENSION A(100)
S     PRINT COMMENT $1$
      READ AND PRINT DATA     N, A, X, EPS
      POL = A(N)
      DER = 0.0
      THROUGH J, FOR I = N-1, -1, I .L. 0
          POL = POL*X + A(I)
          DER = DER*X + (I+1)*A(I+1)
J         CONTINUE
      PRINT RESULTS POL, DER
      H = 1.0
      SLOPE = DER + .ABS. EPS + 1.0
      THROUGH K, FOR M = 0, 1, M.G.28 .OR.
     1           .ABS.(SLOPE-DER).L.EPS
          XH = X + H
          PH = A(N)
          THROUGH L, FOR I = N-1, -1, I .L. 0
              PH = PH*XH + A(I)
    L         CONTINUE
          SLOPE = (PH-POL)/H
          PRINT RESULTS  M, SLOPE
          H = 0.5*H
K         CONTINUE
      TRANSFER TO S
      END OF PROGRAM

$DATA

N=2, A=5.0,-4.0,1.0, EPS=.0001
```

(a)

X=4.0*

| | | | |
|---|---|---|---|
| POL = | 5.000000, | DER = | 4.000000 |
| M = | 0, | SLOPE = | 5.000000 |
| M = | 1, | SLOPE = | 4.500000 |
| M = | 2, | SLOPE = | 4.250000 |
| M = | 3, | SLOPE = | 4.125000 |
| M = | 4, | SLOPE = | 4.062500 |
| M = | 5, | SLOPE = | 4.031250 |
| M = | 6, | SLOPE = | 4.015625 |
| M = | 7, | SLOPE = | 4.007813 |
| M = | 8, | SLOPE = | 4.003906 |
| M = | 9, | SLOPE = | 4.001953 |
| M = | 10, | SLOPE = | 4.000977 |
| M = | 11, | SLOPE = | 4.000488 |
| M = | 12, | SLOPE = | 4.000244 |
| M = | 13, | SLOPE = | 4.000000 |

(b)

X=3.0*

| | | | |
|---|---|---|---|
| POL = | 2.000000, | DER = | 2.000000 |
| M = | 0, | SLOPE = | 3.000000 |
| M = | 1, | SLOPE = | 2.500000 |
| M = | 2, | SLOPE = | 2.250000 |
| M = | 3, | SLOPE = | 2.125000 |
| M = | 4, | SLOPE = | 2.062500 |
| M = | 5, | SLOPE = | 2.031250 |
| M = | 6, | SLOPE = | 2.015625 |
| M = | 7, | SLOPE = | 2.007813 |
| M = | 8, | SLOPE = | 2.003906 |
| M = | 9, | SLOPE = | 2.001953 |
| M = | 10, | SLOPE = | 2.000977 |
| M = | 11, | SLOPE = | 2.000488 |
| M = | 12, | SLOPE = | 2.000244 |
| M = | 13, | SLOPE = | 2.000244 |
| M = | 14, | SLOPE = | 2.000488 |
| M = | 15, | SLOPE = | 2.000977 |
| M = | 16, | SLOPE = | 2.001953 |
| M = | 17, | SLOPE = | 2.003906 |
| M = | 18, | SLOPE = | 2.007813 |
| M = | 19, | SLOPE = | 2.015625 |
| M = | 20, | SLOPE = | 2.031250 |
| M = | 21, | SLOPE = | 2.062500 |
| M = | 22, | SLOPE = | 2.125000 |
| M = | 23, | SLOPE = | 2.250000 |
| M = | 24, | SLOPE = | 2.500000 |
| M = | 25, | SLOPE = | 3.000000 |
| M = | 26, | SLOPE = | .000000 |
| M = | 27, | SLOPE = | .000000 |
| M = | 28, | SLOPE = | .000000 |

(c)

X=2.0*

| | | | |
|---|---|---|---|
| POL = | 1.000000, | DER = | .000000 |
| M = | 0, | SLOPE = | 1.000000 |
| M = | 1, | SLOPE = | .500000 |
| M = | 2, | SLOPE = | .250000 |
| M = | 3, | SLOPE = | .125000 |
| M = | 4, | SLOPE = | .062500 |
| M = | 5, | SLOPE = | .031250 |
| M = | 6, | SLOPE = | .015625 |
| M = | 7, | SLOPE = | 7.812500E-03 |
| M = | 8, | SLOPE = | 3.906250E-03 |
| M = | 9, | SLOPE = | 1.953125E-03 |
| M = | 10, | SLOPE = | 9.765625E-04 |
| M = | 11, | SLOPE = | 4.882813E-04 |
| M = | 12, | SLOPE = | 2.441406E-04 |
| M = | 13, | SLOPE = | 2.441406E-04 |
| M = | 14, | SLOPE = | 4.882813E-04 |
| M = | 15, | SLOPE = | 9.765625E-04 |
| M = | 16, | SLOPE = | 1.953125E-03 |
| M = | 17, | SLOPE = | 3.906250E-03 |
| M = | 18, | SLOPE = | 7.812500E-03 |
| M = | 19, | SLOPE = | .015625 |
| M = | 20, | SLOPE = | .031250 |
| M = | 21, | SLOPE = | .062500 |
| M = | 22, | SLOPE = | .125000 |
| M = | 23, | SLOPE = | .250000 |
| M = | 24, | SLOPE = | .500000 |
| M = | 25, | SLOPE = | 1.000000 |
| M = | 26, | SLOPE = | .000000 |

(d)

Figure 6.33

3. In many cases, functions are given simply by a tabulation of their values at equally spaced values of the independent variable. Let h be the spacing of n + 1 values $x_0, x_2, \ldots, x_{2n}$ and $f(x_0), f(x_2), \ldots, f(x_{2n})$ be the values of the function at these points. The derivative of this function at the midpoints of these intervals (which are labeled $x_1, x_3, \ldots, x_{2n-1}$) may be approximated by

$$f'(x_{2i-1}) \simeq \frac{f(x_{2i}) - f(x_{2i-2})}{h} \qquad \text{for } i = 1, \ldots, n.$$

Using these values the second derivative of the function may be approximated at the original points less the two end points by

$$f''(x_{2i}) \simeq \frac{f'(x_{2i+1}) - f'(x_{2i-1})}{h} \qquad \text{for } i = 1, \ldots, n-1.$$

Similarly, the third derivative may be approximated at the points $x_3, \ldots, x_{2n-3}$. Write a MAD program to read in H, N and the values FN(0), FN(2), ..., FN(2N) of such a function. The program should then compute the first, second and third derivatives and store them in arrays FD, SD, TD. Finally, it should print out a table of values having zeros in the unused places as shown in the example below:

```
     H =  1.000000,      N =         3
FN(0) =  2.000000,  FD(0) =   .000000,  SD(0) =   .000000,  TD(0) =   .000000
FN(1) =   .000000,  FD(1) = -1.000000,  SD(1) =   .000000,  TD(1) =   .000000
FN(2) =  1.000000,  FD(2) =   .000000,  SD(2) =  3.000000,  TD(2) =   .000000
FN(3) =   .000000,  FD(3) =  2.000000,  SD(3) =   .000000,  TD(3) = -1.000000
FN(4) =  3.000000,  FD(4) =   .000000,  SD(4) =  2.000000,  TD(4) =   .000000
FN(5) =   .000000,  FD(5) =  4.000000,  SD(5) =   .000000,  TD(5) =   .000000
FN(6) =  7.000000,  FD(6) =   .000000,  SD(6) =   .000000,  TD(6) =   .000000
```

(Note: Further material on obtaining approximate derivatives in this manner may be found in almost any numerical analysis text under the topic of "finite differences".)

Chapter Seven

# SUBROUTINES

A routine that is designed to be called into use by other routines is referred to as a subroutine. There are two aspects to subroutines: the development or writing of the subroutine itself and the call or use of the subroutine from another routine. Since a single program may involve many such subroutines, the developer of a subroutine must give it a name by which the user may call it. Most subroutine calls also contain a list of actual parameters by means of which the user may specify inputs to the subroutine and other relationships between the subroutine and the rest of the program. These actual parameters must correspond to formal parameters which have been used in writing the subroutine. It is important that both the developer and the user of subroutines understand the relationship between these parameters, so that the connection set up between the called and the calling routines will be correct. If an understanding of this linkage process can be achieved, then the subroutine developer and the subroutine user can work independently. The developer need not know who will use his subroutine, but only precisely what the subroutine is to do and how it is to be called. The user need not know the exact details of how the subroutine goes about its work, but only what data the subroutine needs, what results it produces and how to call it.

In many respects, the development and use of some subroutines is analogous to the development and use of mathematical tables, such as those for logarithms, trigonometric functions or interest charges on loans. It is usually not important to the developer of such a table to know who will use it or what specific problems will be solved with its aid. Correspondingly, the user of such tables seldom knows how they were computed or by whom; he only needs to know what the significance of the values in the table is and how to look them up. For example, using a table which lists values of $\sqrt{x}$ for x running from 0 to 1 in steps of .0001, one may calculate $\sqrt{.5}$, $\sqrt{a^2 + b^2}$ or $\sqrt{2 \cdot \sqrt{c}}$ if the values of a, b, and c are given. In solving problems on a computing machine one seldom uses tables of functions, since a computer can calculate the value of most functions about as quickly as it could look it up in a table. Such tables would also occupy a prohibitively large amount of storage space. Thus, considerable effort is devoted to the development of general subroutines for the common functions.

One of the major advantages of writing algorithms as subroutines is that libraries of the commonly needed ones may be accumulated by individuals and organizations. This practice greatly reduces total programming effort and so it is commonly followed. Most programming systems have numerous subroutines built into them which are automatically supplied if they are called for in a program. Subroutines for most of the common mathematical functions are usually available in such system libraries (see Appendix D, for example). Subroutines for more complicated or less frequently used algorithms are often made available in other ways (for example, on cards) and may thus be shared by many users. To be most useful, such library subroutines should be efficient, convenient to use, and carefully documented.

Even in a single program, the same process may be required in several places, perhaps being applied to different arguments. The use of a subroutine in such instances allows one to describe this process only once and, furthermore, only one area of storage will be occupied by this description. The linkage between calling routine and called subroutine does take both time and storage space, however, and so such subroutines should usually represent a significant amount of work.

By the use of subroutines, a large complex program may be broken down into sections which can then be developed and tested independently. Since the isolation of errors in a routine consisting of several hundred statements may be quite difficult, this technique is often most useful. Care should be taken to logically partition the program into cohesive sections.

Subroutines may usually be defined in two ways. An external subroutine is one which is compiled so that it may be called from any routine. In MAD, such a subroutine is written as part (or all) of an EXTERNAL FUNCTIØN. An internal subroutine (called an INTERNAL FUNCTIØN in MAD) is contained within some larger routine and may only be called in that routine. Similar constructions are available in most other languages. We will normally use the term subroutine to mean external subroutine and specifically use the term internal subroutine when that is meant.

After discussing how existing subroutines may be called, the rest of this chapter is devoted to the methods of defining such subroutines. Internal and external subroutines have nearly the same structure, but the external ones are in some ways more simple and straightforward. Thus, the structure of external subroutines is studied first. A section is then devoted to the details of how the actual parameters of the calling routine are related to the formal parameters of the subroutine. The two following sections give examples which illustrate various structural characteristics of external subroutines and the techniques for writing them. The final section is concerned with the special characteristics of internal subroutines.

## 7.1 The Use of Subroutines

Since the most commonly required subroutines are already written and readily available, this first section is concerned with how to call or use subroutines. In order to illustrate the communication requirements between a calling routine and a subroutine, we will consider two examples.

The first is a subroutine that calculates the positive square root of a number. The input information that must be supplied is the number, the root of which is to be calculated. This number may be given in the form of an arithmetic expression, the value of which is calculated just before entering the subroutine. For example, the calculation of such values as $\sqrt{5}$, $\sqrt{x}$, or $\sqrt{b^2 - 4ac}$ for a given value of each variable is often required. Since functions such as the square root often appear as part of arithmetic expressions, it is convenient to be able to place the call for such a subroutine right in the expression, with the result of executing the subroutine being used in evaluating the expression. For example, one may write an expression essentially equivalent to $(-b + \sqrt{b^2-4ac}) / 2a$ where the square root is understood to mean the positive square root. One further problem that must be considered in this case is what to do if the argument of the square root is negative. It is desirable to build tests into subroutines which will indicate that an error has occurred when given such illegal data. One way of doing this is to simply terminate the user's program with an appropriate message. However, since it is often possible for the programmer to build some remedial action into his calling program, it is helpful to provide some means of returning control to the calling program with an indication that an error has occurred. One way of providing this means is to let the programmer specify the label of a statement in the calling program to which control will be transferred in case an error condition is met.

As a second example, we will consider a subroutine to calculate the solution of a set of simultaneous linear equations. (Such a subroutine will be developed in Chapter 8.) The inputs to this routine must include a reference to the array of coefficients and constants for the equations. As we will see, only the name of such an array or the reference to a vector is used; the structure information is then automatically provided. In addition, the number of equations and the number of unknowns must be supplied to the subroutine. Since this subroutine produces many values, it would not be possible to use its result in the evaluation of an expression in MAD. Thus, a vector in which the results are to be stored is specified to the subroutine and a separate statement is used in the calling program to request the execution of this subroutine.

We have seen an example of a subroutine in each of two classes, one that results in a single value which is used in the evaluation of an expression and another whose call is a simple statement and usually returns several values. Other examples will be discussed later in this section. Subroutines in the first class are called functions, taking their name from the fact that many of these are single-valued mathematical functions. A subroutine in the latter class will be referred to as a procedure. Some functions return values other than the one used in the evaluation of the expression in which they are called. A subroutine of this type will sometimes be referred to as a hybrid subroutine.

In both functions and procedures, a subroutine call consists of a designation of which subroutine is to be used usually followed by a list of parameters enclosed in parentheses. The subroutine designation is usually made by means of a subroutine constant, where the

Definition of a subroutine constant:

<subr-con> is <id> .

All identifiers used in this way (i.e. those followed by a period) in a routine are automatically declared to be in function name mode in that routine by the MAD compiler and thus may not be used for any other purpose. The designation of library subroutines (e.g. SIN., EXP., RANDØM., ATN1.), as well as any subroutines defined by the user, are subroutine constants. Variables, as well as vectors (but not higher order arrays), of function name mode may be used, but these are not followed by a period.

Definition of subroutine variable name, subroutine variable and subroutine designation:

<subr-var-name> is <id>

<subr-var> is <subr-var-name> or <subr-var-name> ( <a-exp> )

<subr-des> is <subr-con> or <subr-var-name> ( <a-exp> ) .

Subroutine variables follow the rules for the formation of other variables. If a vector of subroutine variables is used, it must be dimensioned (unless automatically declared in a VECTØR VALUES statement). Similarly, the mode of such variables must be declared unless they appear in a subroutine designation. An identifier used in a subroutine constant may not also be used in a subroutine variable. Note that a subroutine designation always ends in a period. Also, if a subroutine variable is used in a subroutine description, it must be subscripted; a zero subscript is used if the variable is not a vector, and a dimension statement is not required in this case.

Just as in the case of statement label mode, the variety of expressions of function name mode is somewhat limited. The definition of these expressions does allow the completion of the general definition of an expression which we will need in defining an actual parameter.

Definition of subroutine expression and expression:

<subr-exp> is <subr-con> or <subr-var> or <subr-fn-call>#

<exp> is <a-exp> or <bool-exp> or <label-exp> or <subr-exp>

where <subr-fn-call> is a function name mode function call, which will be defined later in this section.

Using the above definitions a function name substitution statement has the following form:

<subr-var> = <subr-exp>

Although such substitution statements are not frequently used, some rather clever switching of the roles of subroutines may be accomplished with them. For example, the function name mode variable FCN might alternately be assigned the values SIN. and CØS. to determine which of these functions is used. When FCN. is used in a call.

In addition to the subroutine designation, a call usually includes an ordered list of actual parameters which must correspond element by element with a formal parameter list which was used in the definition of the subroutine. That is, each actual parameter is associated with the formal parameter in the same position on the list. Since the subroutine is compiled using the formal parameters and executed using the actual parameters, the corresponding items must agree in mode and usage. Thus, for example, if the formal parameter is in floating point mode, the corresponding actual parameter must also be in floating point mode. Also, if the formal parameter is used on the left of a substitution statement, the actual parameter must be a variable, not a constant or other expression.

Parameters of a subroutine can be used to specify both inputs and outputs. Variables (of any mode), statement labels, references to vectors, and array names may be used as either input or output parameters. In addition, expressions may be used as input parameters. The vector references which may be used as actual parameters are now defined and used in the formal definition of actual parameters.

Definition of a vector reference and an actual parameter:

<vector-ref> is <vector-name> or <vector-name> ( <a-exp> )

<actual-par> is <var> or <exp> or <label> or <subr-des>
or <vector-ref> or <array-name>

The same variable, vector, or array in the actual parameter list may be used as both an input and an output. This can occur in two ways. First, the corresponding formal parameter in the subroutine definition may be used as both an input and an output parameter. The second possibility is that the same variable may occur at two places in the actual parameter list, one of which is an input and the other of which is an output. Such usage can lead to incorrect results if the writer of the subroutine did not take this possibility into account. In most library subroutines where such multiple usage of the same quantity would be useful, it is allowed.

The list of actual parameters may be followed by the label of a statement in the calling program if the subroutine has provision for an error return. This augmented list, referred to as the calling list, is enclosed in parentheses and immediately follows the subroutine designation in the call. If the subroutine has no parameters, the call consists of only the subroutine designation. The formal definition of a subroutine call can now be given as follows:

Definition of subroutine call:

<subr-call> is <subr-des> or <subr-des> ( L{ <actual-par> } )
or <subr-des> ( L{ <actual-par> } , <label> )

The metavariables <fn-call> and <proc-call> may be used to denote a function call and a procedure call, respectively.

A function call may only be used as part of an expression, as indicated in each of the definitions of expressions given earlier. When a function call is included in an expression, the subroutine is executed during the evaluation of that expression. Any expressions occurring as actual parameters in the subroutine call will be evaluated before the subroutine is called and the value returned by the subroutine will be used to continue evaluating the larger expression. In MAD, the value returned by a function has a mode and if this mode differs from the normal mode of the calling routine, then the subroutine name must be declared to be of the correct mode in the calling routine. A prefix to the function call metavariable, <fn-call>, will be used when necessary to indicate the mode of the value returned. For example, <int-fn-call> is an integer mode function call (see definitions in Chapters 4 and 5).

A procedure call is a simple statement which may appear independently or follow the conditional clause in a simple conditional statement. The procedure call statement may have one of two forms,

<proc-call>

or

EXECUTE <proc-call>

which are equally acceptable. Upon the completion of the execution of a procedure, the next statement in the calling routine is executed unless the subroutine encounters some error condition or explicitly provides for some other action.

When a subroutine (function or procedure) provides for an error return and the error condition is encountered during execution, one of two possibilities occurs. If the call for the subroutine contained a statement label following the list of actual parameters (as the final element in its calling list), then control is returned to the statement bearing that label in the calling program. Otherwise, control is passed to the system, an error message is printed and the user's program is terminated. The error message used in this case, as well as other exceptions to the normal return of control to the calling program, will be illustrated in the examples below.

The remainder of this section is devoted to the description of some example subroutines, and to example routines which call these subroutines in various ways.

SQRT.( <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | SQRT.(A), SQRT.(A,ERR) |
| Error condition: | The value of A is negative |
| Value returned: | The positive square root of A |
| Method used: | Newton-Raphson |

A flowchart, the corresponding main routine and the sets of data of Figure 7.1 are designed to illustrate a variety of applications of the call for this function. A hexagonal box is used to denote the subroutine call. When a function is used, as in this routine, the expression may be included in the subroutine box. The normal output of the box (i.e. when no error condition is encountered) is from the flat side at the bottom; the side exits are reserved for error returns. (Flowcharts involving procedure calls will be given later.)

The routine is executed in the following order. After the first set of data is read, the argument of the first call of the function is calculated. It is evaluated as -3.0. Since this is negative, control is transferred to the statement labeled X. The statements computing the square roots of 2.0 and 4.0(Y) and the PRINT RESULTS statement are not executed. The comment is printed, control is transferred to GØ, and the second set of data is read. The argument is calculated for this set of data and found to be equal to 1.0. The expression

(-B + 1.0) / (2.0 * A)

is evaluated and Y1 is set equal to the result, -1.0. The calculations of the square root of 2.0 and of the square root of zero (Y) are made and the resulting

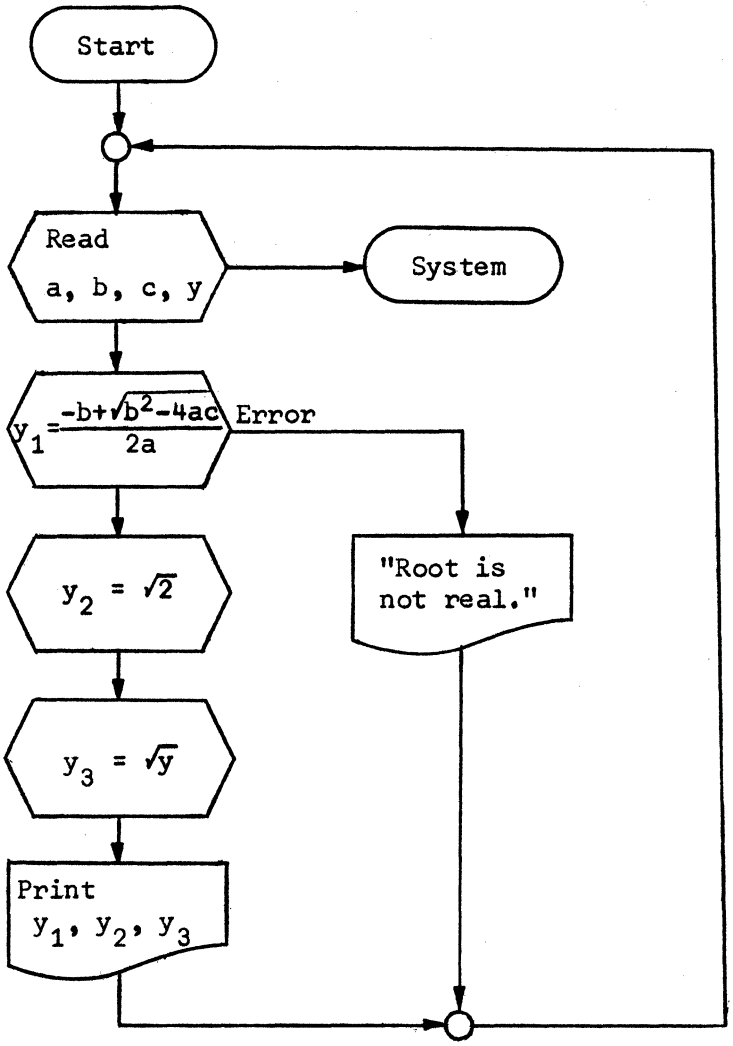



Figure 7.1a

```
GØ          READ AND PRINT DATA    A,B,C,Y
            Y1 = (-B + SQRT.(B.P.2 - 4.0*A*C,X))/(2.0*A)
            Y2 = SQRT.(2.0)
            Y3 = SQRT.(Y)
            PRINT RESULTS Y1, Y2, Y3
            TRANSFER TØ GØ
X           PRINT CØMMENT $ RØØT IS NØT REAL. $
            TRANSFER TØ GØ
            END ØF PRØGRAM
$DATA
A = 1., B = 1., C = 1., Y = 4.*
A = 1., B = 3., C = 2., Y = 0.*
A = 1., B = 2., C = 1., Y = -2.*
A = 2., B = -6., C = 4., Y = 1.*
```

Figure 7.1b

printout is

```
Y1 = -1.000000,   Y2 = 1.414214,   Y3 = .000000
```

Control is again transferred to GØ and the third set of data is read. The argument is found to be equal to 0.0, and Y1 is calculated as -1.0. The square root of 2.0 is next calculated. When an attempt is made to calculate the square root of Y, it is found that Y is negative. Since no statement label is present, control is returned to the system. No further statements in this program are executed. The last set of data is not even read. The comments printed by the system include

```
**** ERROR RETURN TO SYSTEM.
THE FIRST CARD NOT READ DURING EXECUTION WAS --
A = 2., B = -6., C = 4., Y = 1.*
     0 ADDITIONAL CARDS WERE NOT READ
```

An example of a function that has two input parameters, one of which is a vector reference, is a subroutine to calculate the mean value of the components of a vector. This function is not in the MAD library.

MEAN.( <f-p-vector-ref> , <int-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical call: | MEAN.(X,N), MEAN.(X,N,ERR) |
| Error condition: | N is less than or equal to zero |
| Value returned: | The mean of the N components of X, X(0),...,X(N-1). |

Since the first item on the calling list need only be a vector reference, a call that might be used is

```
MEAN.(P(6), 4)
```

where the mean of P(6)...P(9) is calculated.

A third example of a function (also not available in the MAD library) is one to calculate a zero of a function using the half interval method.

HALFIN.( <subr-exp> , <f-p-exp> , <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | HALFIN.(F.,L,R), HALFIN.(F.,L,R,ERR) |
| Restriction: | F. must be a floating point mode function with a single floating point mode parameter which may have an error condition. Thus typical calls of F are F.(X) and F.(X,ERR) |
| Error condition: | F.(L) and F.(R) have the same sign or the error condition of F. is met |
| Value returned: | One zero of F.(X) in the range $L \leq X \leq R$ |
| Method used: | Half interval method, terminating when the interval is less than or equal to .0001 * (R-L) |

A call for this function could be included in a statement such as

```
ZERØ = HALFIN.(SIN., -1., NØZERØ)
```

where the zero of sin x at x = 0 would be found and the value of ZERØ would be set to 0.0. If, on the other hand, the calling statement had been

```
ZERØ = HALFIN.(CØS., A, 0., NØZERØ)
```

where A is -1. or -6.28, the error condition would be met, and the statement labeled NØZERØ would be executed. No value would be established for ZERØ in these cases.

An integer mode function that might prove useful would be one for the factorial (not available in the MAD library).

FACT.( <int-exp> )

| | |
|---|---|
| Subroutine class: | Integer mode function |
| Typical calls: | FACT.(N), FACT.(N,ERR) |
| Error condition: | N is negative |
| Value returned: | N factorial (N!), i.e.<br>N*(N-1)*...*2*1<br>Note 0! = 1 |

The number of combinations of k items from a set of n items, where the items are distinguishable and order is not important (i.e. A followed by B is the same as B followed by A), is given by

$$\frac{n!}{k!\ (n-k)!}$$

This number is always an integer for $n \geq k$. A routine to calculate and print out a list of values for this expression for all values of k up to n, where n is read as input data is as follows:

```
          INTEGER N, K, FACT.
START     READ AND PRINT DATA    N
          THRØUGH S1, FØR K = 1, 1, K .G. N
              CØMB = FACT.(N) / (FACT.(K)*FACT.(N-K))
              PRINT RESULTS K, CØMB
S1        CØNTINUE
          TRANSFER TØ START
          END ØF PRØGRAM
```

Note that the mode of FACT. must be declared; a statement

```
NØRMAL MØDE IS INTEGER
```

could have been used. Also, the statement label for the error condition was omitted since the function is never called with N, K or N-K negative.

Three procedures are available in the MAD library for complex addition, multiplication and division. In these procedures, two floating point numbers are used to represent the real and imaginary parts of each complex number involved. The

first pair of parameters represents the first complex operand, the second pair represents the second operand, and the third pair represents the result. These procedures are described as follows:

CMADD.( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical call: | CMADD.(A,B,C,D,E,F) |
| Error condition: | None |
| Calculations: | E + iF = (A + iB) + (C + iD) |

CMMUL.( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical call: | CMMUL.(A,B,C,D,E,F) |
| Error condition: | None |
| Calculations: | E + iF = (A + iB) · (C + iD) |

CMDIV.( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | CMDIV.(A,B,C,D,E,F)<br>CMDIV.(A,B,C,D,E,F,ERR) |
| Error condition: | C = D = 0.0 |
| Calculations: | E + iF = (A + iB) / (C + iD) |

In the above descriptions, i is used to represent $\sqrt{-1}$. A variable may be used in these subroutines more than once as an input or as both an input and an output. For example,

```
CMADD.(U,V,X,Y,U,V)
CMMUL.(Z,Z,Z,Z,Z,Z)
```

are both legal calls.

A flowchart and the corresponding MAD main routine to calculate the impedance of two complex impedances in parallel, using the formula

$$z = \frac{z_1 z_2}{z_1 + z_2}$$

are shown in Figure 7.2. Note that in the calling lists for CMDIV. and CMMUL. ZR and ZI are used as both inputs and outputs. A zero denominator can occur if both $z_1$ and $z_2$ are zero or if the two impedances are equal in magnitude and of opposite sign (e.g. $z_1 = 2i$, $z_2 = -2i$). Since the parallel impedance is zero in the first case, a test for this condition is made immediately and the computations for this case are treated separately. The second case, where the parallel impedance is infinite, is taken care of with the error return from CMDIV. since this return may occur only in this case.

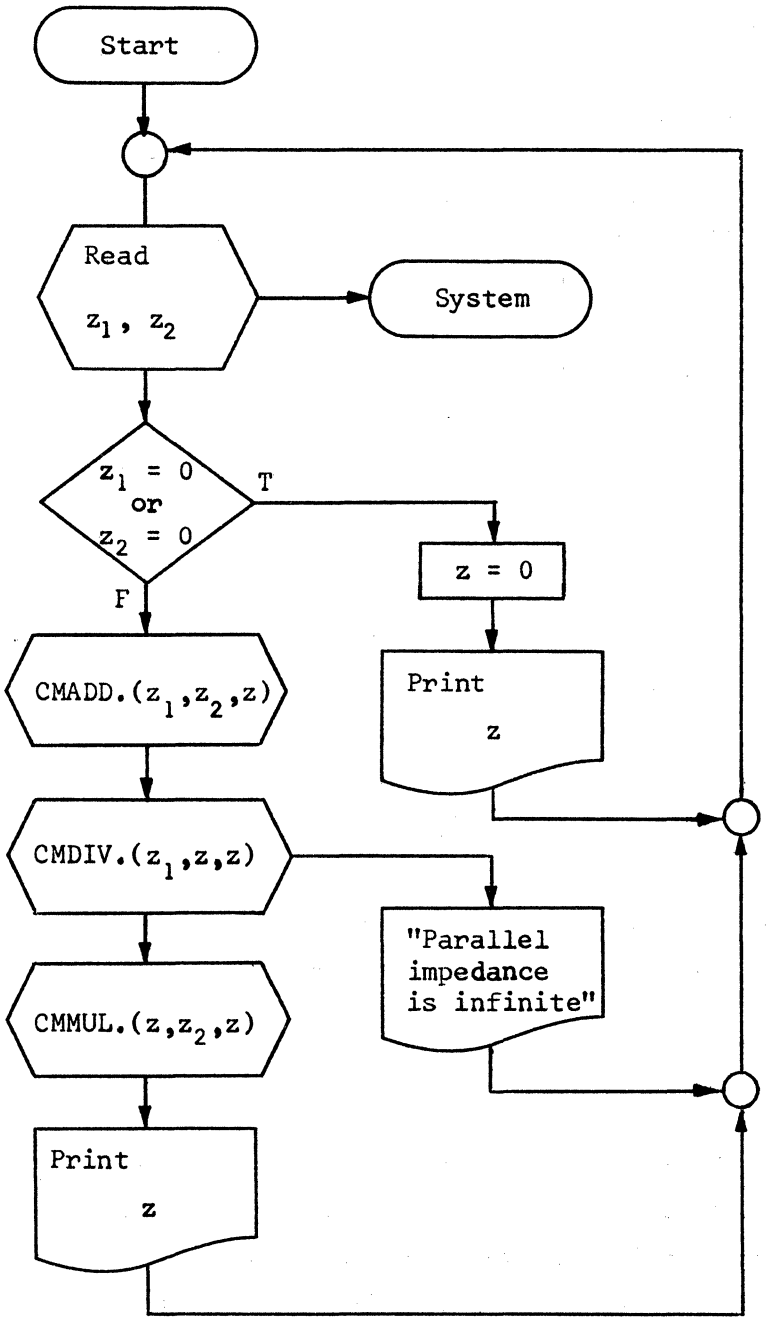

Start
Read
$z_1$, $z_2$
System
$z_1$ = 0
or
$z_2$ = 0
T
F
z = 0
Print
z
CMADD.($z_1$,$z_2$,z)
CMDIV.($z_1$,z,z)
"Parallel
impedance
is infinite"
CMMUL.(z,$z_2$,z)
Print
z

Figure 7.2a

```
          RPARALLEL IMPEDANCE CALCULATIØN
START      READ AND PRINT DATA          Z1R,Z1I,Z2R,Z2I
          RZ1R IS THE REAL PART ØF ØNE IMPEDANCE AND
          RZ1I IS THE IMAGINARY PART ØF THAT IMPEDANCE
           WHENEVER Z1R.E.0. .AND. Z1I.E.0. .ØR. Z2R.E.0. .AND. Z2I.E.0.
               ZR = 0.0
               ZI = 0.0
               PRINT RESULTS ZR, ZI
               TRANSFER TØ START
           END ØF CØNDITIØNAL
           CMADD.(Z1R,Z1I,Z2R,Z2I,ZR,ZI)
           CMDIV.(Z1R,Z1I,ZR,ZI,ZR,ZI,ZERØ)
           CMMUL.(ZR,ZI,Z2R,Z2I,ZR,ZI)
           PRINT RESULTS ZR, ZI
           TRANSFER TØ START
ZERØ       PRINT CØMMENT $ THE PARALLEL IMPEDANCE IS INFINITE.$
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 7.2b

An example of a hybrid subroutine is one named PØLYZ., which is not in the MAD library. It is designed to find a real zero of a polynomial, returning that zero as the value of the subroutine. The subroutine also calculates the coefficients of a reduced polynomial, the input polynomial with the linear term corresponding to the zero factored out. A description of this subroutine is as follows:

PØLYZ.( <f-p-vector-ref>, <f-p-vector-ref>, <int-exp>, <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function modifying other quantities (hybrid subroutine) |
| Typical call: | PØLYZ.(A,B,N,TØL,ERR) |
| Error condition: | The subroutine cannot find a real zero |
| Method used: | A zero z of the n-th order polynomial $p(x) = a_0 + a_1x + \ldots + a_{n-1}x^{n-1} + x^n$ is considered found when $\lvert p(z) \rvert \leq$ TØL. The value returned by the subroutine is z and the coefficients of the (n-1)-th order polynomial p(x)/(x-z) are placed in the vector B. Note: N is unchanged by PØLYZ. and so PØLYZ.(A,A,N,TØL,ERR) is a legal call |

A flowchart of a routine using this subroutine to calculate all the real roots of a polynomial is shown in Figure 7.3a. The subroutine is called successively until either all n roots of the polynomial are found or until PØLYZ. is unable to find another root (whichever occurs first). The index of the loop keeps track of the order of the polynomial. Each time the subroutine is called, the coefficients of the output polynomial are stored again in the first (i-1) components of the input vector. The roots are stored in a vector X. A MAD program for this routine is given in Figure 7.3b.

Another example of a hybrid subroutine is BSL1., the MAD library subroutine for Bessel functions. In this case, the value of the subroutine is used as a flag. It has no numerical significance, but rather indicates whether or not the calculations have been successful.

BSL1.( <f-p-exp>, <int-exp>, <int-exp>, <f-p-var>, <int-exp> )

Subroutine class: Floating point mode function modifying other quantities (hybrid subroutine) Value returned used as flag only

Typical call: BSL1.(X,P,N,B,A)

Error condition: None

Calculations:

X is the argument of the Bessel function (x)

P is a code indicating which Bessel function is to be calculated as follows:

| | |
|---|---|
| 1 | $J_n(x)$ |
| 2 | $I_n(x)$ |
| 3 | $K_n(x)$ |
| 4 | $Y_n(x)$ |
| 5 | $e^{-x}Y_n(x)$ |
| 6 | $e^{-x}K_n(x)$ |

N is the order of the Bessel function (n)

B is the value of the Bessel function when a successful calculation has taken place

A is the accuracy of the result produced where the result contains 27-A bits of accuracy and $0 \leq A < 27$

1 is the functional value returned if the calculations are successful

2 is the functional value returned if the input parameters would produce a result too large for the computer

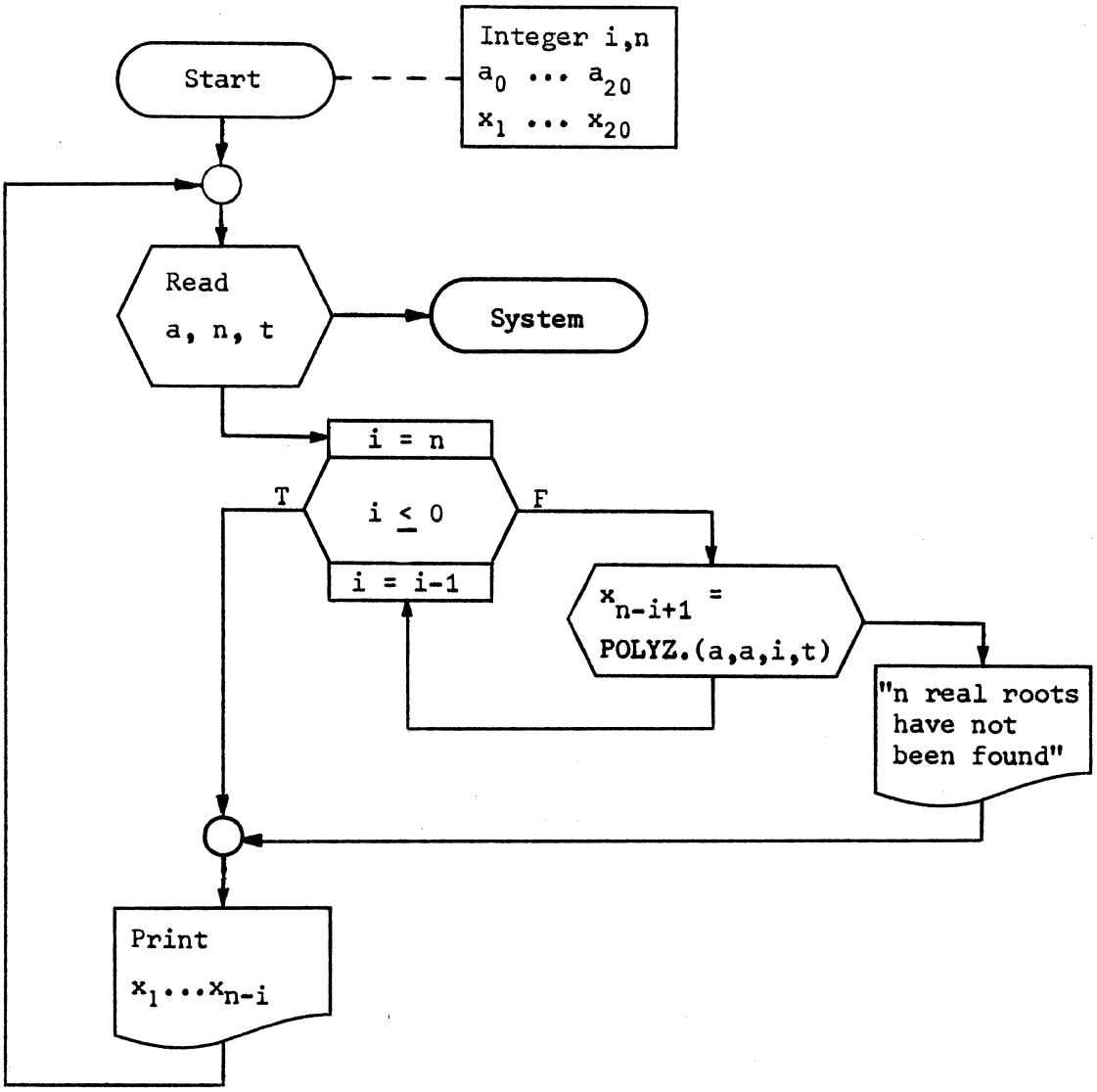



Figure 7.3a

```
          INTEGER I, N
          DIMENSIØN A(19), X(20)
         RPRØGRAM TØ CALCULATE THE REAL RØØTS ØF AN N-TH ØRDER
         RPØLYNØMIAL (N.LE.20) WITH CØEFFICIENTS A(I).
START     READ AND PRINT DATA
          THRØUGH S, FØR I = N, -1, I.LE.0
              X(N-I+1) = PØLYZ.(A,A,I,TØL,ERR)
S         CØNTINUE
S1        PRINT RESULTS X(1)...X(N-I)
          TRANSFER TØ START
ERR       PRINT CØMMENT $ N REAL RØØTS ØF THIS PØLYNØMIAL HAVE NØT BEEN
         1 FØUND.$
          TRANSFER TØ S1
          END ØF PRØGRAM
```

Figure 7.3b

A call for this subroutine could appear in a program in the following set of statements:

```
WHENEVER BSL1.(X,1,1,Y,0) .G. 1.5
    PRINT CØMMENT $ UNSUCCESSFUL BESSEL CALCULATIØN $
    TRANSFER TØ START
END ØF CØNDITIØNAL
PRINT RESULTS Y
```

Here the subroutine call is included in an expression within a relation. When the conditional is executed BSL1. is evaluated and (if the calculations are successful) Y is set equal to $J_1(x)$ in this case. Thus, this segment of program results in either the printing of the comment or the result of the calculation.

As a final example of a subroutine, ANAL. illustrates the use of a statement label vector reference on the parameter list. This routine is not in the MAD library.

ANAL.( <int-exp>, <label> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | ANAL.(N,S), ANAL.(N,S,ERR) |
| Error condition: | N is negative or zero |
| Action taken: | Control is transferred to one of the elements of the statement label vector S dependent on the value of N<br>S or S(0) if N is a prime<br>S(1) if N is a power of a prime<br>S(2) otherwise |

A flowchart of an algorithm to separate the integers from one to one thousand into lists of primes, powers of primes and other non-primes is shown in Figure 7.4a and the corresponding MAD routine is given in Figure 7.4b. Three vectors are constructed. In each, the zeroeth component is used to keep count. Thus P(0) contains the current number of elements in the vector P at any time; and if P(0) is not zero, then P(P) is the last element in the vector.

## Exercises

1. Using the library subroutines for complex arithmetic (described in this section and in the appendix) draw a flowchart and write a MAD main routine to calculate and print out the roots of a quadratic with complex coefficients

$$ax^2 + bx + c$$

where each coefficient is a complex number. For convenience, denote the real part of a by $a_0$ and the imaginary part by $a_1$. Assume $a \neq 0$. The outputs should be $x_0$ (real part of one root) and $x_1$ (imaginary part of that root) and $y_0$ and $y_1$.

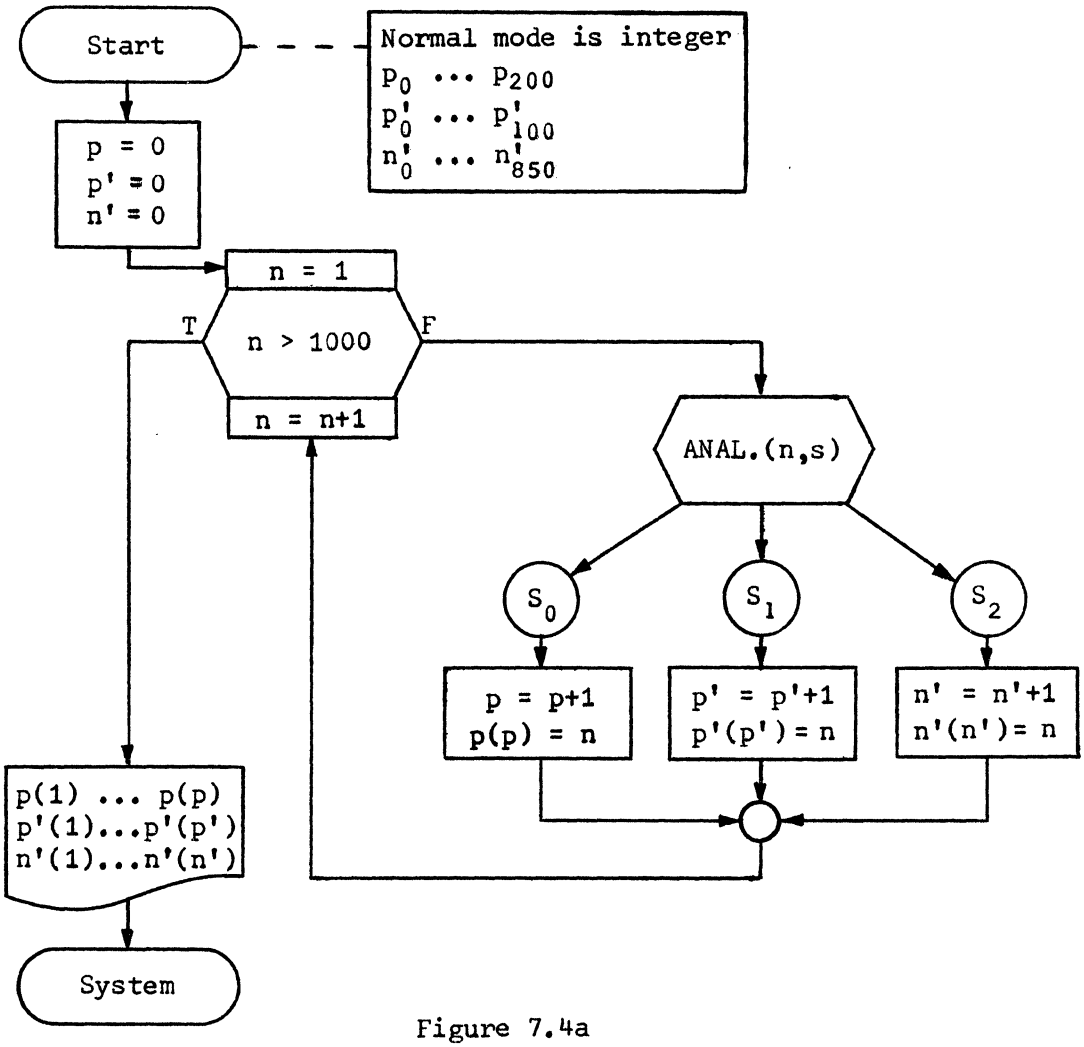



Figure 7.4a

```
                NØRMAL MØDE IS INTEGER
                DIMENSIØN P(200),PP(100),NP(850)
                P = 0
                PP = 0
                NP = 0
                THRØUGH T, FØR N = 1,1,N.G.1000
                    ANAL.(N,S)
S                   P = P + 1
                    P(P) = N
                    TRANSFER TØ T
S(1)                PP = PP + 1
                    PP(PP) = N
                    TRANSFER TØ T
S(2)                NP = NP + 1
                    NP(NP) = N
T               CØNTINUE
                PRINT RESULTS P(1)...P(P),PP(1)...PP(PP),NP(1)...NP(NP)
                END ØF PRØGRAM
```

Figure 7.4b

2. Using the appropriate library subroutines, draw a flowchart and write a MAD main routine to calculate and print out the length of the two sides of a right triangle (in inches), S1 and S2, where the length of the hypotenuse (in inches), H, and one of the angles (in degrees) THETA, are given in the data. Assume H .GE. 0 .AND. THETA .GE. 0 .AND. THETA .L. 90. All inputs and outputs are in floating point mode.

3. Draw a flowchart and write a MAD main routine to read in N + 1 floating point numbers THETA(0)...THETA(N) and the integer N and calculate and print out a four column table, the first column containing $\theta_i$, the second cos $\theta_i$, the third sin $\theta_i$ and the fourth tan $\theta_i$. All angles must be in <u>degrees</u>. Note that the library functions which you must use require angles in radians. Assume all $\theta$ are within the range of the library functions. How do you handle $\theta = 90^o$?

4. The function QUAD. is available to you (Do NOT write a program for it). If C = QUAD.(A,B) (where A is a floating point vector whose components are the real coefficients of the polynomial equation $a_0 + a_1x + a_2x^2 = 0$ and the value of QUAD. is in integer mode) is used in a program, the following results are produced:

1) If the roots of the equation are real and equal, C is set equal to zero, and B(0) is the value of the root.
2) If they are real and unequal, C is set equal to 1 and the roots are B(0) and B(1).
3) If they are imaginary, C is set equal to 2 and the roots are B(0) and B(1).
4) If they are complex, C is set equal to 3 and B(0) is the real part of the first root, B(1) is the imaginary part of the first root, B(2) is the real part of the second root and B(3) is the imaginary part of the second root.

Write a <u>complete</u> MAD main routine that reads A as input data and, using QUAD. for the calculations, prints out the <u>non-zero</u> quantities

XR1 = the real part of root 1
XI1 = the imaginary part of root 1
XR2 = the real part of root 2
XI2 = the imaginary part of root 2

Those of the four quantities that are zero must not be printed except when there is a zero real root. As an example, the printout for $x^2 + 4 = 0$ would be

XI1 = 2.000000
XI2 = -2.000000

and that for $x^2 - 2x = 0$ would be

XR1 = .000000
XR2 = 2.000000

## 7.2 Definition of External Subroutines

The structure of algorithms is substantially the same whether they are implemented as main routines or as subroutines. However, in most languages used today (including MAD) these two are treated quite differently. The differences involve the initiation of the execution, the handling of inputs and outputs, and what follows the completion of execution. Main routines in MAD are initiated by a user's request (the EXECUTE on the control card), whereas subroutines are called by other routines. Inputs and outputs to subroutines may be communicated between the calling routine and the subroutine by means of parameters, as well as by use of the same type of input and output statements as in main routines. Subroutines are normally terminated by returning to the routines that called them, whereas main routines return to the system. In some sense the main routine is thus a subroutine to the system. In fact, it would be possible for a language to use only subroutines and this approach will be discussed in Section 7.4.

It must be possible for the MAD compiler to distinguish between subroutines and main routines, and it is best that it be able to do this immediately when it begins work on the source language for a routine. To make this possible, the first statement of an external subroutine is always in one of the following four forms:

```
EXTERNAL FUNCTIØN  <subr-con>  ( L{ <formal-par> } )
EXTERNAL FUNCTIØN  ( L{ <formal-par> } )
EXTERNAL FUNCTIØN  <subr-con>
EXTERNAL FUNCTIØN
```

The list of formal parameters, if any are used in the subroutine, must be included in this statement. One entry point to the subroutine may be indicated in this statement by including the subroutine constant by which it is designated. Formal parameters and entry points will be discussed in detail in this section. The compiler determines that a main routine is being defined by the fact that such routines do not begin with an EXTERNAL FUNCTIØN statement.

A definition of formal parameters and a discussion of how they are associated with the actual parameters in a call is now in order.

Definition of formal parameter or dummy name:

```
<formal-par>  is  <var-name>  or  <label-name>
                  or  <subr-var-name> .
```

Note that only names are allowed as formal parameters, not expressions, not constants, not subscripted variables and not labels. The declaration of names as

formal parameters is accomplished by putting them in the formal parameter (or dummy name) list in the first statement of the external function. The mode of formal parameters must be declared with the usual mode declarations unless their mode is declared automatically or they are to have the normal mode. Formal parameters must not be dimensioned and must not appear in vector values declarations.

During compilation of a subroutine, a formal parameter table is established which contains the mode for each formal parameter used in the routine. This table is in addition to the name table that is formed for all other names used in the subroutine. The mode, dimensioning and vector initialization of other names are declared in the same way as in main routines. The tables developed for these names, as well as for constants, are of the same format as those in main routines. The mode declared for the formal parameters (or normal mode) is used along with these tables to construct the object code. The program is then compiled in the same manner as a main program except that references to the location of formal parameters cannot be filled in. A parameter use table is produced during compilation, listing all references to formal parameters in the machine code. The size of this table is limited, and if this maximum is exceeded, a diagnostic of the form

**** PARAMETER USE TABLE EXCEEDED

is printed. The number of references to formal parameters can often be reduced by setting a temporary variable equal to a formal parameter immediately upon entry to the subroutine and using this variable in place of the formal parameter. When this is not possible, the number of entries in this table can be reduced by use of PRØGRAM CØMMØN (see Appendix F). As a final step in the compilation the parameter use table is used to generate object program which will insert the address of the corresponding actual parameters into each instruction involving a parameter. This insertion is the first thing which occurs when the subroutine is executed. Each insertion requires time and thus the efficiency of the routine may be improved by reducing the number of references to formal parameters. The address inserted may be that of a variable, a constant, or a temporary location if the actual parameter is a composite expression. The address of a transfer to a subroutine or of a statement label may also be inserted (See Section 6.4). Thus the storage assigned to the actual parameters in the calling routine is used by the subroutine. When the reference is to a vector or an array, the structural information about the actual parameter is used by the subroutine and so no dimensioning declaration is required for formal parameters (in fact, such declarations must not be given).

Each actual parameter must be in the same mode as the corresponding formal parameter. The subroutine and calling routine are compiled independently, and thus

it would not be possible for the subroutine to even detect that the mode of the calling parameter differs from that of the formal parameter. If, for example, the formal parameter were declared to be in integer mode, the corresponding actual parameter would be interpreted as an integer. If it were not in integer mode, erroneous results would be produced.

If a formal parameter appears on the left of a substitution statement, then a value is established for that variable and the storage location in the calling routine is modified. This value is thus available to the calling routine and may be obtained by a reference to the calling name corresponding to this formal parameter. Note, however, that if the calling item is a constant, the storage location of the constant is referred to and the value of the constant is modified by the subroutine. Similarly, if the calling item is an expression, a temporary location which is not addressable by the calling program is modified and thus the results are not available to the calling program. Thus, if a dummy variable is modified by the subroutine, i.e. if it appears on the left of a substitution statement, the corresponding item in the calling list must be a variable. One should avoid modifying an input parameter unless that parameter is also used as an output. Otherwise, the corresponding calling parameter will no longer be available to the calling routine.

The entry point to a subroutine (i.e. the point where execution begins) may be indicated by a subroutine entry statement of the form:

```
ENTRY TØ  <subr-con>
```

This statement allows the subroutine constant to be associated with any point in the definition. Statements preceding the entry statement may be executed only if a transfer to them occurs. If execution of a subroutine is to begin with the first executable statement in the definition, its subroutine constant may follow EXTERNAL FUNCTIØN in the first statement. Thus, for example,

```
EXTERNAL FUNCTIØN ABC.(X,Y,Z)
```

is equivalent to the two statements

```
EXTERNAL FUNCTIØN (X,Y,Z)
ENTRY TØ ABC.
```

The first statement to be executed when a subroutine has been called is the subroutine entry statement which causes the locations of the actual parameters to be substituted in those instructions which refer to formal parameters.

When more than one subroutine uses the same formal parameter list, their definitions may be combined. This is particularly useful when the algorithms involved have substantial parts in common. Each such subroutine requires an entry statement

containing its designation and indicating the point in the definition at which its execution begins. Subroutine entry statements which are encountered in executing such a definition are skipped over by automatically inserted transfer statements and so it is not necessary for the programmer to insert such transfers. Since any subroutine entry to a combined definition will cause actual parameter address substitutions to be made for all formal parameter references used in the definition, it is not economical to combine definitions just because they have identical formal parameter lists. Such combinations should be made only if a major part of the programs for the various subroutines being defined are shared in common.

The normal return from a subroutine to the calling program is produced by the function return statement which has one of the following forms:

```
FUNCTIØN RETURN
FUNCTIØN RETURN  <exp>
```

The first form is used for procedures where the results are returned using the parameter lists. Thus, all that is necessary in this case is to return to the calling program. In the case of functions (including hybrid subroutines), a value must be associated with the function upon return to the calling program and this value is just the value of the expression included in the second form of the function return statement. The mode of the function is determined by the mode of this expression.

In subroutines which test for error conditions, some means must be provided for an error return to the calling program. The error return statement is simply

```
ERRØR RETURN
```

and this statement is often included in a simple conditional which tests for the error condition. If an error return statement is executed and in the call for this subroutine an error statement label was included at the end of the calling list, control is returned to the calling program at the statement bearing that label. If no such label was included in the calling list, control is returned to the system when an error return statement is executed and the system then terminates the user's program. There is never a statement label (or anything else) on the formal parameter list to correspond to the label used for an error return on the calling list. Although more than one error return statement may be included in the same definition, only one will be executed in any call. All error return statements return to the same statement in the calling program and so it is not possible to obtain returns to different statements in the calling program using error return statements. One may, however, include one or more labels in the formal and actual parameter lists, and transfer to these under various circumstances. This technique will be discussed later in the chapter.

The last statement of every subroutine definition must be

END ØF FUNCTIØN

which is normally used only as an indication to the compiler that the end of the subroutine definition has been reached. This statement is executable, but it returns control to the system for termination of the user's program. Thus, both the normal returns and the error returns to the calling program must be explicitly provided in the routine.

A subroutine may call another subroutine that is defined in a separate program (or an internal function, defined within the same program). Without special precautions (see Appendix F), a subroutine may not call itself or any other subroutine defined in the same subroutine definition.

## 7.3 An External Function Definition

In order to illustrate the definition of external functions, we now present an example of a simple function definition, namely the function MEAN. of Section 7.1 which calculates the arithmetic mean, or average, of the first n components of a vector a.

MEAN.( <f-p-vector-ref> , <int-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical call: | MEAN.(X,N), MEAN.(X,N,ERR) |
| Error condition: | N is less than or equal to zero |
| Value returned: | The mean of the N components of X, X(0),...,X(N-1). |

A flowchart and the corresponding MAD subroutine are shown in Figure 7.5.

The first statement of the subroutine contains the declaration of the formal parameters which are used. Remarks which describe the subroutine in general terms follow, since it is good practice to include such descriptive remarks at the beginning of all subroutines. Other remarks may be included within definitions where needed. As in a main program, the normal mode is assumed to be floating point unless otherwise declared. Mode declarations must be included for all names not in the normal mode and thus N and I are declared to be integers. A dimension declaration must not be used for the vector A since it is a formal parameter. The declarations (other than the formal parameter declaration) may be included anywhere within the program.

The next statement is the entry statement, which indicates the starting point of the subroutine having MEAN. as its designation. This is the first statement executed when MEAN. is called and it causes the locations of the actual parameters to be substituted for the formal parameters. In this definition the entry could have been indicated in the first statement by writing.

```
EXTERNAL FUNCTIØN MEAN.(A,N)
```

since no executable statements precede the entry point. The next statement which is executed when MEAN. is called is the test for the error condition. This test is made with a simple conditional clause which controls execution of the error return statement. The next four statements in this routine, through the one labeled S, are a set of typical statements to calculate the sum of N numbers, whether in a main routine or a subroutine. Since this routine is a function, the next statement serves two purposes. It indicates at what point in the execution of the subroutine a return is to be made to the calling program and contains an expression whose value becomes the value of the function.

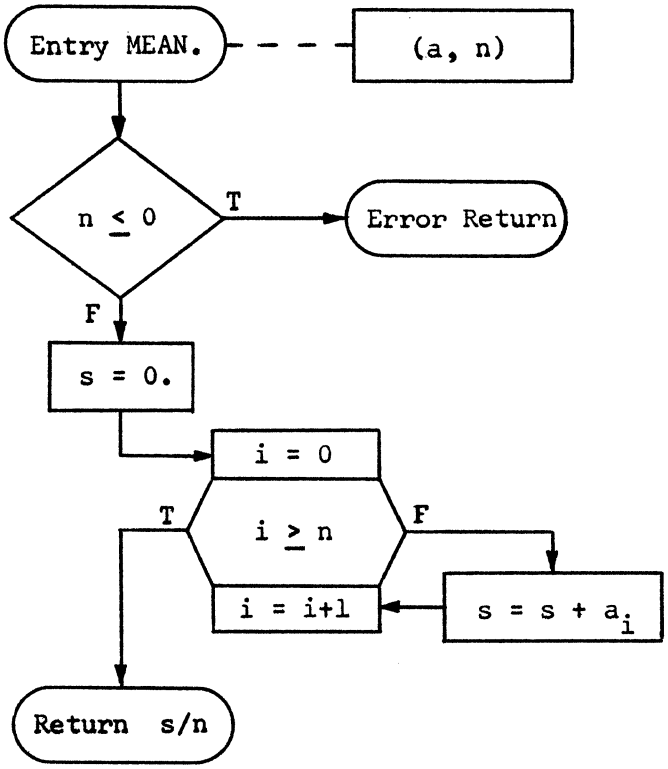



Figure 7.5a

```
      EXTERNAL FUNCTIØN (A,N)
     RFUNCTIØN TØ CALCULATE THE MEAN ØF N FLØATING
     R     PØINT MØDE NUMBERS A(0)...A(N-1)
      INTEGER N, I
      ENTRY TØ MEAN.
      WHENEVER N.LE.0, ERRØR RETURN
      SUM = 0.0
      THRØUGH R, FØR I = 0, 1, I.GE.N
           SUM = SUM + A(I)
R     R    CØNTINUE
      FUNCTIØN RETURN SUM/N
      END ØF FUNCTIØN
```

Figure 7.5b

Note that the flowchart and the MAD program for this algorithm are nearly the same as those for an algorithm designed to be a main program. The major differences are as follow:

1) There are no read statements, the input data being transmitted through the parameter lists.
2) The output information is transmitted by the two exits, not printed directly in the subroutine.
3) The subroutine makes the calculations for only one set of data and then returns to the calling program.

Although most subroutines have these features, it is possible to read and print in a subroutine, as well as loop on several sets of data.

A flowchart and the corresponding MAD statements for a main routine that utilizes MEAN. are shown in Figure 7.6. Before discussing the storage allocation and the parameter lists, let us examine the two calls for MEAN. which appear in this routine. The first call calculates the mean of the 10 components of Y, Y(0)...Y(9). Since a constant number of values is always involved, this constant (10) is used as the actual parameter, and the label for the error condition is omitted. In the second call, the mean of a variable length vector is found where the length of that vector is read as input data. This call results in the calculation of the mean of the N components of X, X(1)...X(N). If N is less than or equal to zero, the error return is executed and only the result of the first call is printed (since no value was established for M2).

Even though the same name is used in both the subroutine and the calling program (e.g. N and R), no connection is made between these names. The programs are compiled separately and thus names in each are treated independently. The only correspondence that is made is between items in the same place in the actual and formal parameter lists. The same meaning is attached to a name in a subroutine and the calling routine only if that name appears in the same position on both the calling list and the dummy name list (e.g. N).

Some tables, similar to those established by the MAD compiler for this main program and subroutine are shown in Tables 7.1 and 7.2. They have been modified so as to simplify their presentation, but the basic idea remains unchanged. The memory locations are the actual addresses (in octal).

The locations of the actual parameters which are substituted for the formal parameters on the first call are shown in Table 7.3a. Thus, a reference to A(0) causes the number from location 10170 (which is X(0)) to be used, a reference to A(1) takes the number from location 10167 (which is X(1)), and so forth. A reference

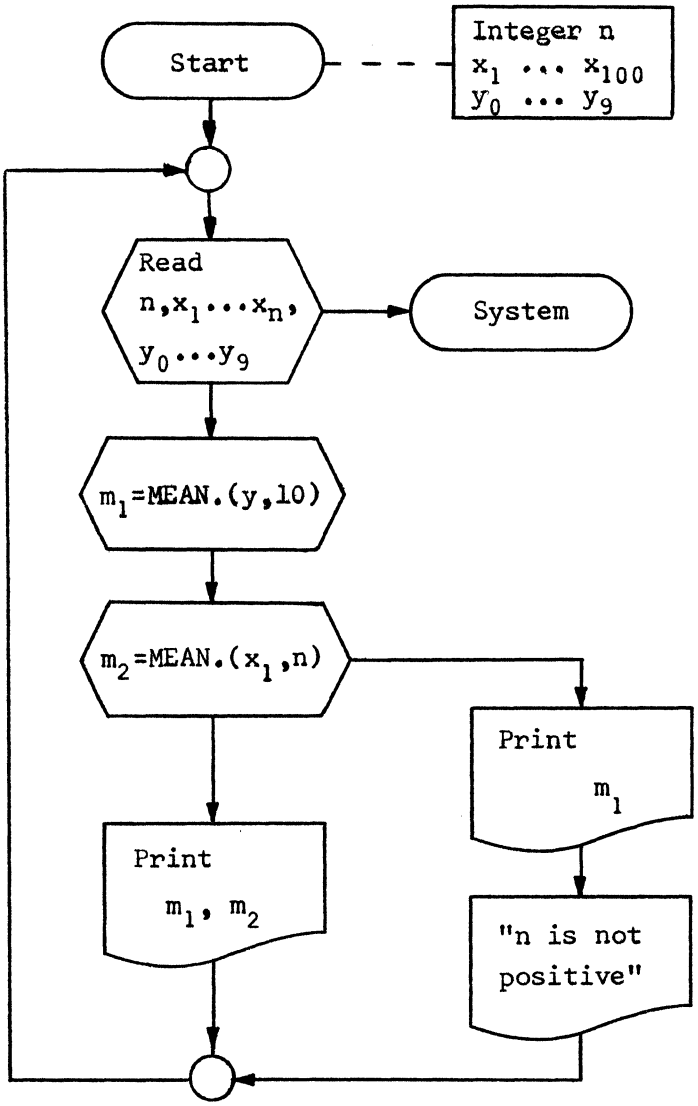



Figure 7.6a

```
          RRØUTINE TØ ILLUSTRATE THE USE ØF MEAN.
           INTEGER N
           DIMENSIØN X(100), Y(9)
START      READ AND PRINT DATA      X,N,Y
           M1 = MEAN.(Y,10)
           M2 = MEAN.(X(1),N,R)
           PRINT RESULTS M1, M2
           TRANSFER TØ START
R          PRINT RESULTS M1
           PRINT CØMMENT $ N IS NØT PØSITIVE. $
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 7.6b

Table 7.1 Main Routine Storage Assignments

| Constant | Mode | Location |
|---|---|---|
| 1 | Integer | 10171 |
| 10 | Integer | 10172 |

| Name | Mode | Size | Location |
|---|---|---|---|
| M1 | Floating point | 1 | 10005 |
| M2 | Floating point | 1 | 10006 |
| N | Integer | 1 | 10007 |
| R | Label | 1 | 10010 |
| START | Label | 1 | 10011 |
| X | Floating point | 101 | 10156 |
| Y | Floating point | 10 | 10170 |

Table 7.2 Subroutine Tables

a. Storage Assignments

| Constant | Mode | Location |
|---|---|---|
| 0 | Integer | 10256 |
| 0.0 | Floating point | 10257 |
| 1 | Integer | 10260 |

| Name | Mode | Size | Location |
|---|---|---|---|
| I | Integer | 1 | 10253 |
| R | Label | 1 | 10254 |
| SUM | Floating point | 1 | 10255 |

b. Formal Parameter Table at Compilation

| Formal Parameter | Mode |
|---|---|
| A | Floating point |
| N | Integer |

Table 7.3 Parameter Association at Execution

a. First call - MEAN.(Y,10)

| Formal Parameter | Mode | Actual Parameter | Location |
|---|---|---|---|
| A | Floating point | Y | 10170 |
| N | Integer | 10 | 10172 |

b. Second call - MEAN.(X(1),N,R)

| Formal Parameter | Mode | Actual Parameter | Location |
|---|---|---|---|
| A | Floating point | X(1) | 10155 |
| N | Integer | N | 10007 |

to N in the subroutine takes the number from location 10172 (where the constant 10 is stored).

To emphasize the importance of being consistent in the mode of corresponding items on the two parameter lists, let us see what happens if the call were

```
M1 = MEAN.(Y, 10.0)
```

The only change in these tables is that the mode of the constant 10.0 (Table 7.1) becomes floating point and thus it is stored as +204500000000. When the subroutine is called, it is this number, interpreted as an integer, that is used for N in the subroutine. Thus, N is interpreted as approximately $2^{34}$ (or, in decimal, a number greater than $10^{10}$).

The locations used for the formal parameters in the second call for MEAN. are shown in Table 7.3b. For this call, a reference to A, draws the number from location 10155, the location reserved for X(1) in the main program. Also, A(1) refers to location 10154, to X(2), and so forth.

Exercises

1. a) Draw a flowchart and write a MAD external function definition for the following subroutine:

LGST.( <f-p-vec-ref>, <int-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical call: | LGST.(X,N), LGST.(X,N,ERR) |
| Error condition: | N less than or equal to zero |
| Value returned: | The value of this function is largest element of the N+1 components of the vector X, X(0)...X(N) |

b) Write a MAD main routine to test LGST..

2. a) Modify LGST., calling it in INTSML., so that it returns the smallest element of an integer mode vector. The value of LGST. is thus in integer mode.

b) Write a MAD main routine to test INTSML..

## 7.4 Other External Subroutine Definitions

An example of a procedure definition is one for complex multiplication which behaves the same as CMMUL. (the library subroutine). A flowchart and the corresponding MAD subroutine are shown in Figure 7.7.

This subroutine demonstrates two important points. First, since it is a procedure, the normal return is just

```
FUNCTIØN RETURN
```

without any expression. Second, the use of a temporary variable G is required to allow the same calling variable as an input and output, e.g. a call such as

```
CMMULT.(W,X,Y,Z,W,X)
```

Table 7.4 illustrates the pertinent portion of the storage table from a calling program containing this call and shows the correspondence between the actual parameters and the formal parameters established by this call. Let us consider what would happen if the three arithmetic statements used in the subroutine were replaced by

```
E = A*C - B*D
F = A*D + B*C
```

For the call shown, the value of the expression to the right of E would be calculated correctly and stored in the corresponding actual parameter W at location 10006. But now, when F is calculated, A is obtained from W at location 10006, which now contains the value of E, the real part of the product, not the real part of the first operand. F would therefore be assigned an incorrect value.

In general, when the same calling variable, vector or array may be used in two or more places on the calling list, one of which is an output, care must be taken in the function definition to preserve all inputs until they are needed. Temporary variables for some of the outputs and/or some of the inputs are usually defined for this purpose.

As an example of a combined definition of two functions, we will describe functions to approximate the area under a curve using two different approximation (See Section 6.7 for discussion of methods). The information about the function is given in a vector Y, the k-th component of which gives the value of the function at $x = a + kh$. A zeroeth order approximation to the area in the interval from $a + kh$ to $a + (k + 1)h$ is $hy_k$ and the corresponding approximation (left sum) to the total area from a to $a + nh$ is given by

$$I_o = h \sum_{k=0}^{n-1} y_k$$

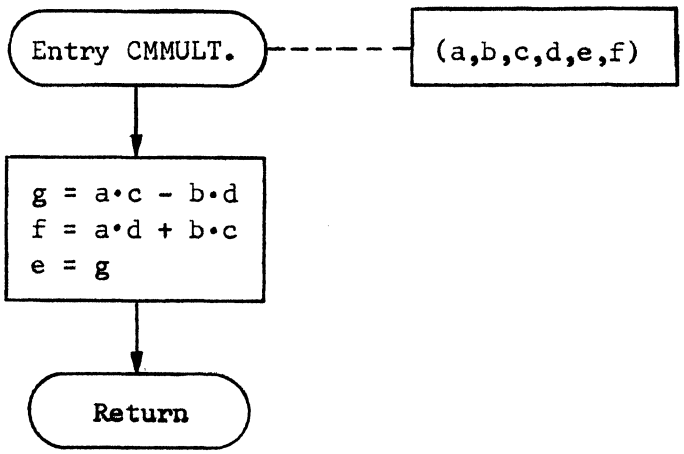



Figure 7.7a

```
 EXTERNAL FUNCTIØN (A,B,C,D,E,F)
RPRØCEDURE TØ ØBTAIN E + I * F = (A + I * B) * (C + I * D)
R    WHERE I = (-1).P..5
 ENTRY TØ CMMULT.
 G = A * C - B * D
 F = A * D + B * C
 E = G
 FUNCTIØN RETURN
 END ØF FUNCTIØN
```

Figure 7.7b

Table 7.4 Parameters used in CMMULT.(W,X,Y,Z,W,X)

a. Calling Routine Storage Assignments

| Name | Mode | Size | Location |
|---|---|---|---|
| W | Floating point | 1 | 10006 |
| X | Floating point | 1 | 10007 |
| Y | Floating point | 1 | 10010 |
| Z | Floating point | 1 | 10011 |

b. Parameter Association at Execution

| Formal Parameter | Mode | Actual Parameter | Location |
|---|---|---|---|
| A | Floating point | W | 10006 |
| B | Floating point | X | 10007 |
| C | Floating point | Y | 10010 |
| D | Floating point | Z | 10011 |
| E | Floating point | W | 10006 |
| F | Floating point | X | 10007 |

The first order approximation approximates the area under the curve by a series of trapezoids, and the area is thus given by

$$I_1 = \tfrac{1}{2} h \sum_{k=0}^{n-1} (y_k + y_{k+1})$$

$$= \tfrac{1}{2} h (y_o + y_n) + h \sum_{k=1}^{n-1} y_k$$

A flowchart of an algorithm to define the two subroutines, designated AREA0. and AREA1., is shown in Figure 7.8a and the corresponding MAD coding is given in Figure 7.8b.

In this combined definition, the value of S is initialized to one of two values, depending upon the entry. The same loop is then used for the remainder of the calculations, with the same return used for either call.

The formal parameter list of a subroutine may include one or more dummy subroutine names. The corresponding items on the calling list must then be references to subroutines available in the calling routine, including library subroutines, other external functions included in the same set of routines, and internal functions that are part of the calling program. The dummy subroutine and the calling subroutine must be of the same type and must have corresponding sets of parameters.

As an example of this type of subroutine, let us consider a modification of AREA1. that calculates the area, using the trapezoidal rule, of a single valued function of a single floating point mode parameter. A description of this subroutine is

AREAF1. ( <subr-des> , <int-exp> , <f-p-exp> , <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | AREAF1.(F.,N,A,B), AREAF1.(F.,N,A,B,ERR) |
| Restriction: | F. must be a floating point mode function with a single floating point mode parameter and may have an error condition |
| Error condition: | N is less than 1 or the error condition of F. is satisfied |
| Value returned: | This function calculates the trapezoidal approximation of the area of f(x) from x = a to x = b, using n equal intervals |

A flowchart for this function is shown in Figure 7.9a and the corresponding MAD subroutine definition in Figure 7.9b.

Any reference in this subroutine to the dummy function name will result in a call for the actual function, the name of which appears on the calling list. The loop is terminated when x is greater than b-.5h, and not when x equal b, because

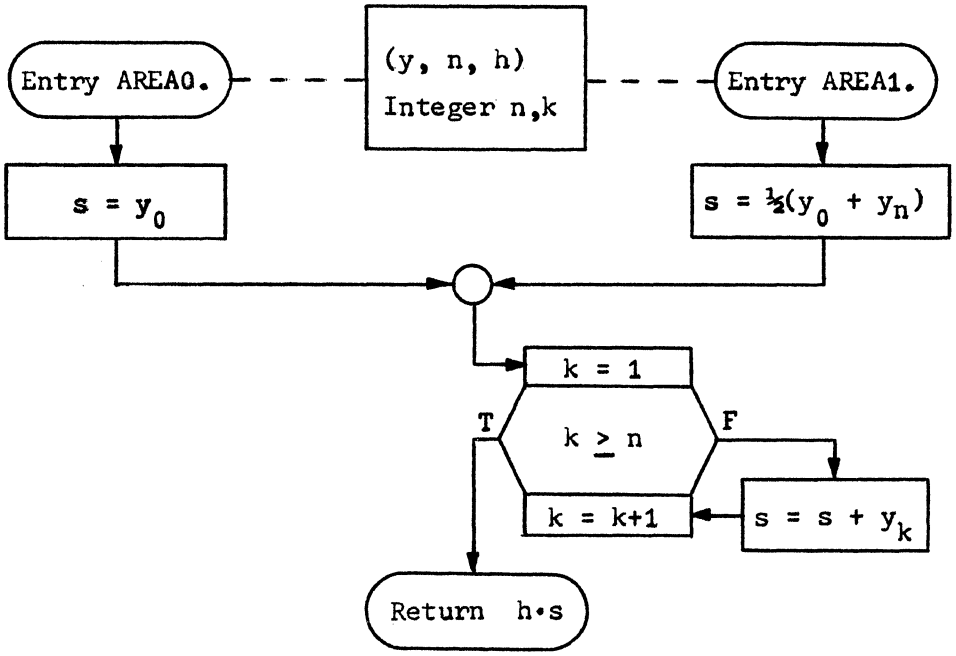



Figure 7.8a

```
                EXTERNAL FUNCTIØN (Y,N,H)
                INTEGER N,K
               RFUNCTIØN TØ CALCULATE ZERØETH ØRDER APPRØXIMATIØN
               R    TØ THE AREA FØR N + 1 EVENLY SPACED PØINTS.
                ENTRY TØ AREA0.
                S = Y(0)
                TRANSFER TØ SUM
               RFUNCTIØN TØ CALCULATE FIRST ØRDER APPRØXIMATIØN
               R    TØ THE AREA FØR N + 1 EVENLY SPACED PØINTS.
                ENTRY TØ AREA1.
                S = 0.5*(Y(0) + Y(N))
SUM             THROUGH S1, FOR K = 1,1,K.GE.N
                    S = S + Y(K)
S1              CØNTINUE
                FUNCTIØN RETURN H*S
                END ØF FUNCTIØN
```

Figure 7.8b

of the roundoff problem. That is, when h is added to a the required n times, the sum may be slightly less than or slightly greater than b. Thus, a test on $x \geq b$ may result in the addition of an extra term, namely that for x just less than b. The variable y is introduced so that the expression b-.5•h need be calculated only once, not each time through the loop. A main routine to call AREAF1. and a test function to be used with these routines are shown in Figure 7.9c (the three routines must be included in one program). Note that in each of the calls for F., a label E is included for the error condition of the subroutine corresponding to F. (e.g. CØS.). The label E must appear in AREAF1., the routine in which the call appears. Thus, in order to return to the main routine, the label E is placed on an error return statement.

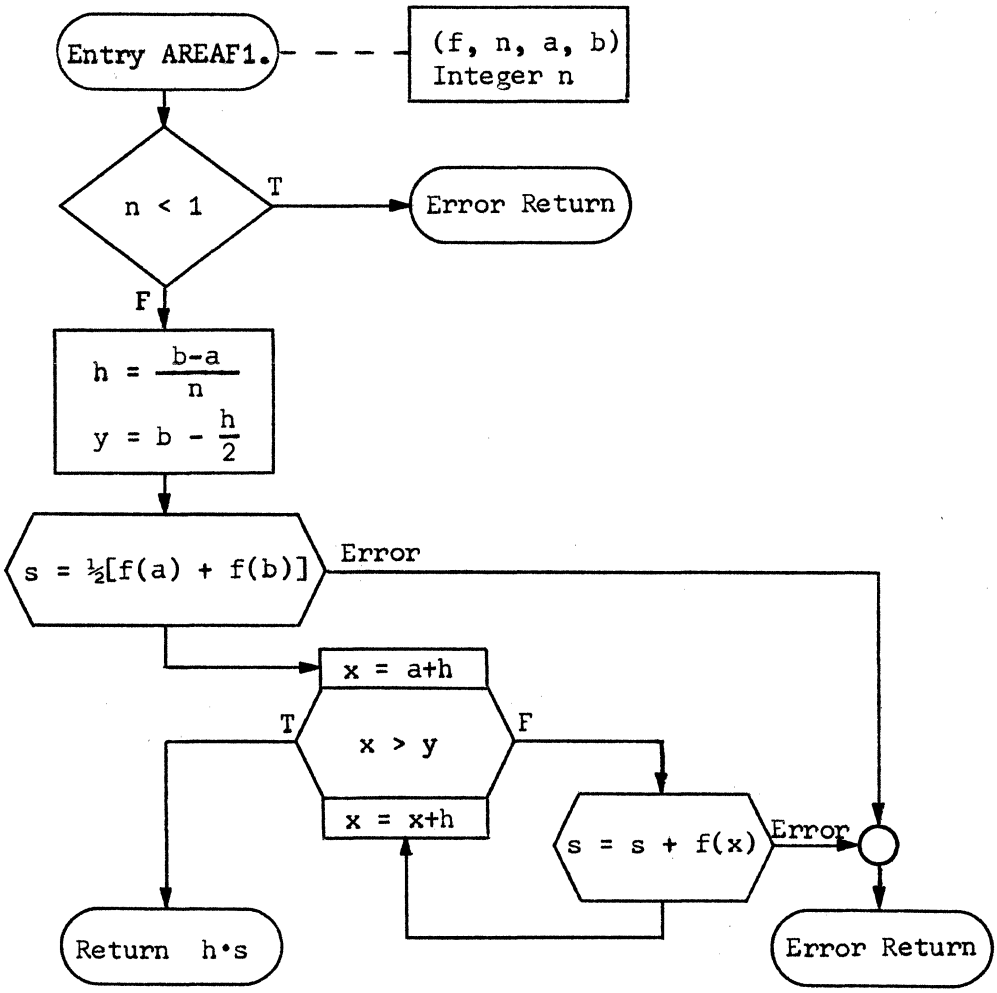



Figure 7.9a

```
      EXTERNAL FUNCTIØN (F.,N,A,B)
     RFUNCTIØN TØ CALCULATE TRAPEZØIDAL APPRØXIMATIØN
     R    TØ THE AREA ØF F.(X) FRØM A TØ B USING
     R    N EQUAL INTERVALS.  F. MUST BE A FLØATING
     R    PØINT MØDE FUNCTIØN WITH A SINGLE FLØATING
     R    PØINT MØDE PARAMETER.
      INTEGER N
      ENTRY TØ AREAF1.
      WHENEVER N.L.1, ERRØR RETURN
      H = (B-A)/N
      Y = B - .5*H
      S = .5*(F.(A,E) + F.(B,E))
      THRØUGH S1, FØR X = A + H, H, X.G.Y
          S = S + F.(X,E)
S1    CØNTINUE
      FUNCTIØN RETURN H*S
E     ERRØR RETURN
      END ØF FUNCTIØN
```

Figure 7.9b

```
ACØS = AREAF1.(CØS.,0.,1.5707963,100)
AREA = AREAF1.(TEST.,0.,1.,50)
PRINT RESULTS ACØS, AREA
END OF PRØGRAM

EXTERNAL FUNCTIØN(X)
ENTRY TØ TEST.
FUNCTIØN RETURN 5. + X*(6. + X*(3. + X*(2. + X)))
END ØF FUNCTIØN
```

Figure 7.9c

The first call for AREAF1. calculates the area of cos x from x = 0 to x = $\pi/2$. On the first call of AREAF1., the references to F. in the definition of AREAF1. result in a call for the library subroutines CØS.. The test program also includes a subroutine called TEST., whose value is that of the polynomial $x^4 + 2x^3 + 3x^2 + 6x + 5$. This subroutine is called whenever F. is referenced during the second call of AREAF1. The name of any subroutine of the proper form included in the program or the library may be used in the calling list.

A subroutine variable name without a period may be used on the formal parameter list also. However, it must be followed by a subscript and a period in a subroutine designation. Thus, the declaration in Figure 7.9b could be modified to

```
EXTERNAL FUNCTIØN (F,N,A,B)
```

but the computations involving F must also be modified as follows:

```
S = .5*(F(0).(A,E) + F(0).(B,E))
S = S + F(0).(X,E)
```

An example of a hybrid subroutine is one that calculates the roots of a quadratic. Its description is as follows:

QUADRT.( <f-p-vec-ref>, <f-p-vec-ref> )

| | |
|---|---|
| Subroutine class: | Integer mode function, returning other values (hybrid subroutine) |
| Typical calls: | QUADRT.(A,B), QUADRT.(A,B,ERR) |
| Error condition: | $a_2$ equals zero. |
| Computations: | This subroutine calculates the roots of a quadratic equation $a_2x^2 + a_1x + a_0 = 0$. If the roots are real, the value of QUADRT. is 0 and the roots are given in $b_0$ and $b_1$. If the roots are complex (including imaginary), the value of QUADRT. is 1 and $b_0$ is the real part of both roots and $b_1$ is the magnitude of the imaginary part. (The roots are $b_0 + ib_1$ and $b_0 - ib_1$.) |

A MAD subroutine for QUADRT. is given in Figure 7.10a, the corresponding flowchart is shown in Figure 7.10b, and a MAD main routine to test this subroutine is shown in Figure 7.10c. The algorithm is a simplified version of the one presented in Chapter 4.

```
 EXTERNAL FUNCTIØN (A,B)
RHYBRID SUBRØUTINE TØ CALCULATE THE RØØTS ØF
R    A(2)*X*X + A(1)*X + A(0) = 0.
RVALUE IS INTEGER MØDE, EQUAL TØ ZERØ IF RØØTS
R    ARE REAL, ØNE ØTHERWISE
RB(0) IS FIRST REAL RØØT ØR REAL PART ØF CØMPLEX RØØT
RB(1) IS SECØND REAL RØØT ØR MAGNITUDE ØF IMAGINARY PART
 ENTRY TØ QUADRT.
 WHENEVER A(2).E.0., ERRØR RETURN
 Q = A(1)*A(1) - 4.*A(2)*A
 WHENEVER Q.L.0
     B = -.5*A(1) / A(2)
     B(1) = .ABS.(.5*SQRT.(Q) / A(2))
     FUNCTIØN RETURN 1
 END ØF CØNDITIØNAL
 Q = SQRT.(Q)
 B = (-A(1) + Q)*.5 / A(2)
 B(1) = (-A(1) - Q)*.5 / A(2)
 FUNCTIØN RETURN 0
 END ØF FUNCTIØN
```

Figure 7.10a

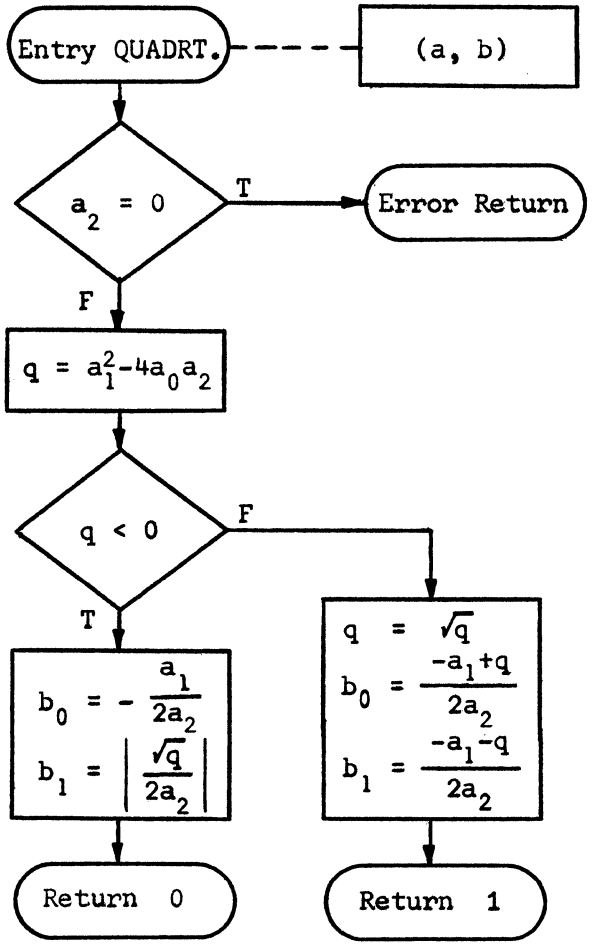



Figure 7.10b

```
          RRØUTINE TØ TEST QUADRT.
           DIMENSIØN A(2),X(1)
           INTEGER QUADRT.
START      READ AND PRINT DATA        A(0)...A(2)
           WHENEVER QUADRT.(A,X).E.0
               PRINT CØMMENT  $OREAL RØØTS $
               PRINT RESULTS X,X(1)
           ØTHERWISE
               PRINT CØMMENT  $OCØMPLEX RØØTS,X(0) IS REAL PART,
          2X(1) IS IMAGINARY PART. $
               PRINT RESULTS X,X(1)
           END ØF CØNDITIØNAL
           TRANSFER TØ START
           END ØF PRØGRAM
```

Figure 7.10c

In this subroutine definition, information is returned to the calling program both through the parameter list (the value of the roots) and through the FUNCTIØN RETURN statement (the form of the roots). Since the latter information is in integer mode, the function name must be declared to be in integer mode in the calling routine. Two separate FUNCTIØN RETURN statements appear in this routine, at most one of which is executed in any call for this subroutine. A library subroutine for the square root is utilized in this subroutine. The variable Q is defined and modified so as to reduce the calculation time required by the computer. Since A and B are formal parameters, no dimension declaration is used in the subroutine definition. However, the calling vectors, A and X must be dimensioned in the calling routine. Note that the only call for QUADRT. in the main routine is in the arithmetic expression that is part of a relation. When this relation is evaluated, the subroutine is called and thus values of X(0) and X(1) are calculated. It is not necessary that QUADRT. appear in a substitution statement.

As a final example of a subroutine definition, let us consider the definition of ANAL., a procedure to determine whether an integer is a prime, a power of a prime, or neither. The description of this subroutine, first presented in Section 7.1, is as follows:

ANAL.( <int-exp> , <label> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | ANAL.(N,S), ANAL.(N,S,ERR) |
| Error condition: | N is negative or zero |
| Action taken: | Control is transferred to one of the elements of the statement label vector S dependent on the value of N<br>S or S(0) if N is a prime<br>S(1) if N is a power of a prime<br>S(2) otherwise |

A flowchart for this subroutine is given in Figure 7.11a. After eliminating the integers less than or equal to zero as errors and treating 1, 2 and 3 as special cases, the smallest proper factor of n (if any exists) is found. If the number can be factored, the smallest factor must be a prime number, less than or equal to $\sqrt{n}$. All prime numbers greater than two are odd and thus, after examining two, all odd numbers are tested as possible factors of n. Although less tests would be required if only the primes were tested, the work involved in determing whether a potential factor is a prime would be greater than the savings obtained.

If a factor, i, is found, then n is divided by i successively to determine whether or not n is a power of i. This subroutine does not return any information about the factors of n or, if n is a power of a prime, what that prime is. Modifica-

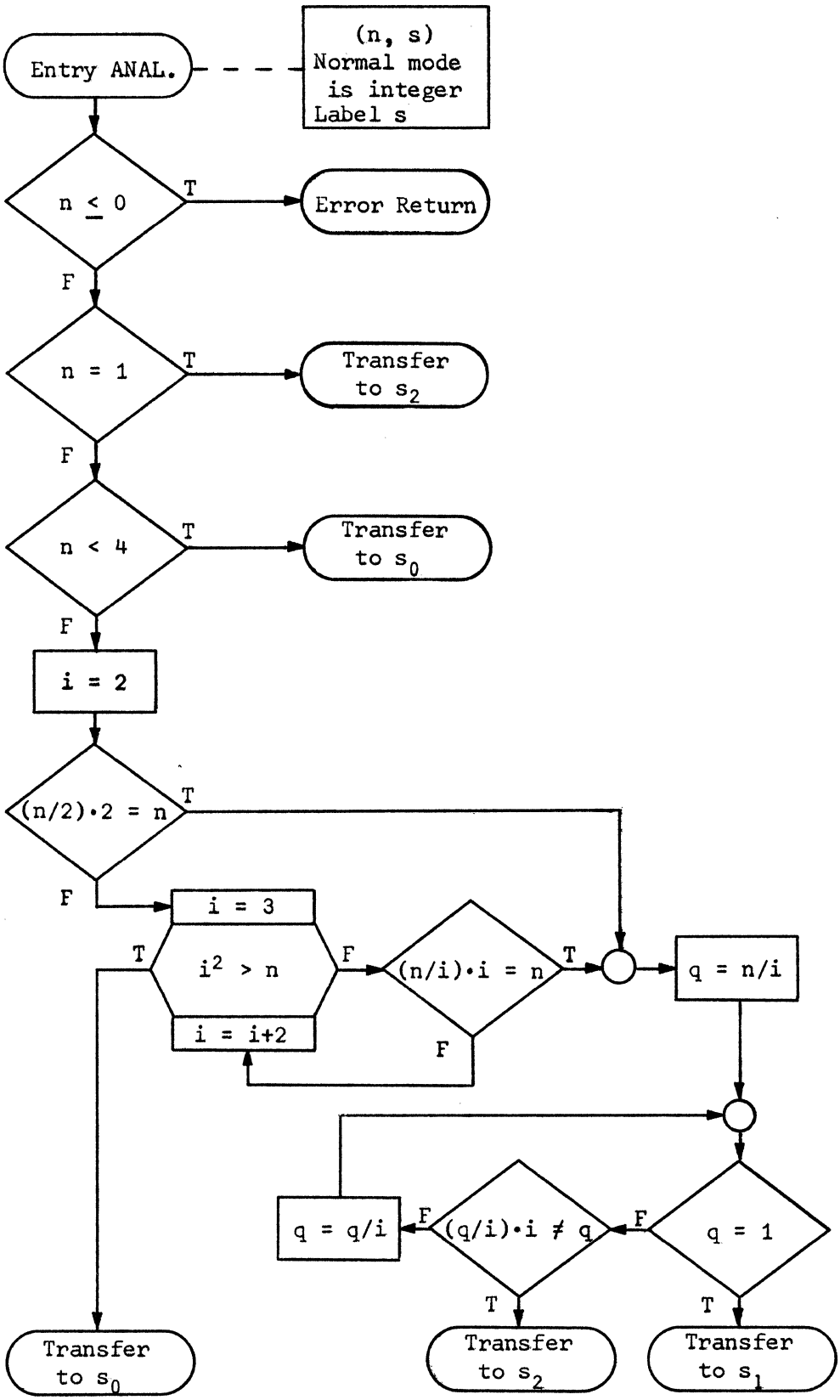

Entry ANAL.
(n, s)
Normal mode
is integer
Label s
n ≤ 0
T
Error Return
F
n = 1
T
Transfer
to $s_2$
F
n < 4
T
Transfer
to $s_0$
F
i = 2
(n/2)•2 = n
T
F
i = 3
T
$i^2 > n$
F
(n/i)•i = n
T
i = i+2
F
q = n/i
q = q/i
F
(q/i)•i ≠ q
F
q = 1
T
T
Transfer
to $s_0$
Transfer
to $s_2$
Transfer
to $s_1$

Figure 7.11a

tions of ANAL. to perform these tasks are suggested in the exercises. A MAD routine for this subroutine is shown in Figure 7.11b. When a transfer statement is executed, and the label in the transfer statement is a dummy label, control is transferred to the calling program--to the statement carrying the corresponding label. Since the dummy label does not appear in the statement label columns (1-10), it is necessary to declare that the mode of this name is statement label.

Transfer returns may occur in subroutines that have normal FUNCTIØN RETURN statements (either functions, procedures, or hybrid subroutines). The transfer exit can be used to modify the sequential execution of the calling program. It may also be used to provide transfers to different locations in the calling program under different error conditions.

Transfer returns may be chained as follows: A subroutine, say X., has a transfer return to a dummy statement label. The routine calling that subroutine is also a subroutine, e.f. Y.. Furthermore, the label in the call for X. is a dummy label of Y. and thus when the transfer exit is executed in X., control is returned not to Y., but to the routine calling Y.. This can be extended indefinitely.

```
      EXTERNAL FUNCTIØN (N,S)
     RPRØCEDURE TØ DETERMINE WHETHER N IS PRIME (S(0)),
     R    A PØWER ØF A PRIME (S(1)) ØR NEITHER (S(2)).
     REXIT IS ØN A TRANSFER STATEMENT
      NØRMAL MØDE IS INTEGER
      STATEMENT LABEL S
      ENTRY TØ ANAL.
      WHENEVER N.LE.0, ERRØR RETURN
      WHENEVER N.E.1, TRANSFER TØ S(2)
      WHENEVER N.L.4, TRANSFER TØ S(0)
      I = 2
      WHENEVER (N/2)*2.E.N, TRANSFER TØ T1
      THRØUGH T2, FØR I = 3, 2, I*I.G.N
          WHENEVER (N/I)*I .E. N, TRANSFER TØ T1
T2    CØNTINUE
      TRANSFER TØ S(0)
T1    Q = N/I
T3    WHENEVER Q.E.1, TRANSFER TØ S(1)
      WHENEVER (Q/I)*I.NE.Q, TRANSFER TØ S(2)
      Q = Q/I
      TRANSFER TØ T3
      END ØF FUNCTIØN
```

Figure 7.11b

As mentioned in Section 7.2, it would be possible to treat main routines as subroutines and thus eliminate the need for distinguishing between these different kinds of routines. We now indicate how this might have been done in MAD. Taking any MAD program, its main routine could be converted to a subroutine by adding the statement EXTERNAL FUNCTIØN MAIN. at the beginning (assuming that MAIN. is not otherwise used as an identifier) and replacing the END ØF PRØGRAM statement with an END ØF FUNCTIØN statement. A trivial main routine consisting of the statements

```
EXECUTE MAIN.
END ØF PRØGRAM
```

could then be added and the program would operate essentially as before. Although the system could have been designed to automatically supply this two statement main routine, there would be better ways of handling the situation. First, it would be desirable to allow the user to designate his "main" routine as he pleases. For example, he might like to call it BØSS., TØPDØG. or some other exotic name. Second, the control card specification EXECUTE is already an implied call of the main routine. It would be possible to have the user specify on the control card the designation of the first subroutine to be executed. For example

```
$ EXECUTE BØSS.
```

would specify the routine designated BØSS. as the first to be executed. This arrangement would be consistent with the other control specifications such as

```
$ CØMPILE MAD
```

which are actually calls on subroutines of the operating system. Although these features might be very useful, they are <u>not</u> presently available in most systems, including those which contain MAD.

<u>Exercises</u>

1. Modify the definition of Figure 7.8b so that an explicit transfer around the second entry is not needed. In addition to deleting the transfer, only one statement need be modified.

2. Modify ANAL., calling the new subroutine ANAL1., such that the calling list includes, following the two items of ANAL., an integer mode output variable, the value of which is the smallest factor of N (which may be N itself if N is prime); and a second integer mode output variable, the value of which is the power of that factor.

3. Using ANAL1., draw a flowchart and write a MAD main routine to calculate and print out all the factors of a positive integer N. If the input is N = 252, the output should be of the form

```
N(0)...N(5)
252      2      2      3      3      7
```

4. a) Draw a flowchart and write a definition in MAD for the following external function using the algorithm described in Section

HALFIN.( <subr-ref> , <f-p-exp> , <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | HALFIN.(F.,L,R), HALFIN.(F.,L,R,ERR) |
| Restriction: | F. must be a floating point mode function with a single floating point mode parameter which may have an error condition. Thus typical calls of F are F.(X) and F.(X,ERR) |
| Error condition: | F.(L) and F.(R) have the same sign or the error condition of F. is met |
| Value returned: | One zero of F.(X) in the range $L \leq X \leq R$ |
| Method used: | Half interval method, terminating when the interval is less than or equal to .0001 * (R-L) |

b) Write a main routine to call HALFIN. and a subroutine TEST.(X), the value of which is x(sin x + 2 cos x). The interval and the value of EPS are to be read as data.

5. a) Draw a flowchart and write an external function definition for

NEWRAF.( <subr-ref> , <f-p-exp> , <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical call: | NEWRAF.(F.,XO,EPS,ERR) |
| Error condition: | A root of F.(X) was not found in 25 iterations starting from XO, a zero slope was encountered, or overflow occurred |
| Value returned: | The value of this function is a root of F.(X) found using the algorithm described in Section 6.5. |

b) Write a main routine to call NEWRAF. and a subroutine TEST1.(X), the value of which is cosh x - 10. (Note: cosh $x = .5(e^{x} + e^{-x})$ and is not available in the library). In the main routine, use XO = 1 and EPS = $10^{-4}$.

## 7.5 Internal Subroutine Definitions

Up to this point, we have been discussing the definition of external subroutines. In this section, we will discuss the definition of internal subroutines, called internal functions in MAD.

The major difference between internal and external subroutine definitions is that internal subroutines are contained in the definition of other routines. They are compiled simultaneously with those routines, and thus all names used in internal subroutines have the same meaning (and use the same storage locations) as the same name in the remainder of that routine. This allows inputs and outputs of internal subroutines to be conveyed both through the parameter lists (in exactly the same way as in external subroutines) and through variables appearing in both the internal subroutine and the rest of the routine. Formal parameters of an internal subroutine may not be used elsewhere in the routine of which that subroutine is a part.

An internal function definition may occur anywhere in a routine except within another internal function. Its location does not affect the order of execution of the routine. Declarations for an internal function (same requirements as for an external function) may be placed anywhere in the routine of which the internal function is a part.

The name of an internal subroutine has meaning only in the routine in which it is defined. This name may be a calling parameter of a subroutine called from that routine and this is the only way that an internal subroutine may be used outside the routine in which it is defined.

Two structures are available for internal subroutine definitions in MAD. The first is similar to that for external subroutines. It begins with one of the statements

```
INTERNAL FUNCTIØN  <subr-con>  ( L{ <formal-par> } )
INTERNAL FUNCTIØN  ( L{ <formal-par> } )
INTERNAL FUNCTIØN  <subr-con>
INTERNAL FUNCTIØN
```

and, as in the case of an external function, ends with a statement

```
END ØF FUNCTIØN
```

The statements in the body of the definition follow the same rules as those in an external subroutine.

The second structure which has no external subroutine counterpart is known as the one statement internal function. Any function, the value of which can be specified by a single expression, may be defined in this way. The general form of such

a definition is

INTERNAL FUNCTIØN <subr-con> ( L{ <formal-par> } ) = <exp>

where the mode of the expression determines the mode of the function. Mode declarations for formal parameters, and for the subroutine name may appear anywhere in the routine. Examples of one statement internal function definitions are

```
INTERNAL FUNCTIØN TEST.(X) = 3. + X*(4. + X)
INTERNAL FUNCTIØN X.(A,B) = A*SIN.(B) + A(1)*CØS.(B)
INTERNAL FUNCTIØN REM.(N,P) = N - N/P*P
INTERNAL FUNCTIØN T.(C,D) = C.G.0. .ØR. (C-D).P.2. .G. 1.
```

In the second example, a subscripted dummy variable is used; it is not dimensioned. Also, library subroutines for the sine and cosine are called. In general, subroutines that are in the library or are part of the program as well as other internal functions in the same routine, may be called by an internal function. In the third example, the dummy variables are in integer mode and must be so declared in the routine of which this internal function is a part. Also, the mode of T. is Boolean and this must be so declared.

Finally, we will consider an example of a main routine to calculate, the mean of a set of n numbers, utilizing an internal function to calculate the sum. The MAD routine is shown in Figure 7.12. (Note: the labels L1 and L2 are introduced only to aid the discussion below; they are not needed by the routine.)

Two key points to note are (1) the placement of the internal function within the calling routine and (2) the use of the variable N. The location of the internal function is arbitrary. After the execution of the statement labeled L1, the next statement executed is the one labeled L2. The variable N is used both in the

```
          DIMENSIØN A(1000)
          INTEGER N,I
S2        READ AND PRINT DATA       A(1)...A(N),N
L1        MEAN = SUM.(A) / N
          INTERNAL FUNCTIØN (X)
          ENTRY TØ SUM.
          S = 0
          THRØUGH S1, FOR I = 1, 1, I.G.N
          S = S + X(I)
S1        CØNTINUE
          FUNCTIØN RETURN S
L2        PRINT RESULTS MEAN
          TRANSFER TØ S2
          END ØF PRØGRAM
```

Figure 7.12

internal function and the rest of the routine. The same storage location is used for N in both places and thus N is interpreted as the same number in both places.

When writing internal functions, care must be taken to avoid using the same name that is used elsewhere in the routine unless the same meaning is intended. This applies to label names as well as variable names, and, thus, if S1 is used as a label in the internal function, it must not be used elsewhere in that routine.

Exercises

1. State which of these MAD function calls are proper, and correct those that are not. Describe the error, in each case of error. Where a library function name is used, make sure that its use is correct.

a) TANX = SIN.(X)/CØS.(X)

b) Y = A(0)*LN.(1. / 2*PI*R)

c) B = 1.414*CØS.(650)

d) Z = CMMUL.(A,B)

e) MURDER.

f) BØY.(W,I,T) = 2*J

g) CØS.(2*PI*F*T) = 0.5

h) The following are to be considered as a consecutive pair of statements in the same program

W = 41°

B = 2*SIN.(W)

2. Answer each of the questions true or false. Explain what is wrong with those to which the answer is false.

a) The number of items on the calling list must always be the same as the number of formal parameters in the function definition.

b) Subscripted variables may never be used in the formal parameter list of a function definition.

c) The last statement of an external subroutine (before END ØF FUNCTIØN) must be FUNCTIØN RETURN.

d) The statement FUNCTIØN RETURN must always be followed by a list of variables (one or more), the values of which are the results of the function.

e) A function call may never appear in a substitution statement.

3. A certain external function begins with the statements:

```
EXTERNAL FUNCTIØN (X,Y,Z,W)
INTEGER W
FLØATING PØINT S,X,Y,Z
ENTRY TØ EXTEN.
WHENEVER W.E.0, ERRØR RETURN
S = X(1) + X(2) + X(3) + X(4)
```

In a main routine this external function is to be called in the statement

```
Y = EXTEN.(A,4.0,B + C,X)
```

Answer the following questions as completely as you can on the basis of just the above information.

a) What DIMENSIØN statement(s) are required in the main program?

b) What variables must be declared INTEGER in the main program?

c) What will happen if the calling statement is obeyed with X = 0?

d) What DIMENSIØN statement(s) are required in the external function?

e) What restriction(s) on the use of Y and Z in the external function must be adhered to for this call to be meaningful?

4. Assume that A and B are floating point, C and D are integer, E and F are Boolean, and G and H are statement label mode. The first statement in each case below is a procedure call in a main routine and the second is the first statement of the definition of the procedure called. Are these statements compatible and if not, why not?

a)
```
GØHØME.(A,B,C,D)
EXTERNAL FUNCTIØN  GØHØME.(B,A,C,D)
```

b)
```
TØNITE.(A - 3*B,B(3),G)
EXTERNAL FUNCTIØN  TØNIGHT.(B,A)
```

c)
```
TRUBLE.(C)
EXTERNAL FUNCTIØN  TRUBLE.(C + D - 1)
```

d)
```
MATRIX.(A(1),A(2),A(3),C)
EXTERNAL FUNCTIØN  MATRIX.(B(1),B(2),B(3),C)
```

e)
```
MIXED.(A,B,C)
EXTERNAL FUNCTIØN  MIXED.(C,B,A)
```

5. a) Write an external function TIME., the dummy variable of which is the annual interest rate in decimal (.06 in place of 6 percent), that calculates and returns the number of years it takes for money to double when the interest is compounded quarterly.

b) Write a main program that

a) reads and prints data
b) calls the function
c) prints the answer

This program may be very short.

6. One way of handling complex numbers in a computer is to use vector notation, where a complex number A could be written in cartesian form and would be stored with A(1) as the real part and A(2) as the imaginary part. A(0) is being reserved for part d. Another way that a number could be stored is in polar form with A(1) as the magnitude and A(2) as the angle in degrees. For some applications, it is more convenient to use the first notation, (e.g. addition) and for others, the second (e.g. multiplication).

a) Write three external subroutine definitions, each having three formal parameters, e.g. A, B, C, which are vector representations of three complex numbers. The first subroutine should be designated CVADD. for addition which calculates C = A + B with all variables in cartesian form; the second, CVMPY. for multiplication which calculates C = A*B with all variables in polar form, and the third, CVDIV. for division which calculates C = A/B with all variables in polar form. In both CVMPY., and CVDIV., the output angle C(2) should be reduced so that $-180^{\circ} < C(2) \leq 180^{\circ}$. Since this reduction will be useful in other subroutines, it would be most convenient to implement it in a separate external subroutine.

b) To use the routines of part (a), it is necessary to convert from polar form to cartesian form and from cartesian form to polar form. Write two external subroutine definitions using two formal parameters each, e.g. A, B, which are vector representations of complex numbers, to perform the conversions. The procedure designated CART. should convert a number in polar form to cartesian form and one designated PØLAR. should do the opposite. In writing your procedure, be sure it can handle the calls CART.(X,X). and PØLAR.(X,X).

c) Use the definitions of parts a and b in a main routine that calculates and prints out in polar form the impedance of the parallel combination of two impedances, each given as data in cartesian form. The equations are given on page 271.

d) Use A(0) to store a code character which indicates whether A is in polar or cartesian form. Modify your definitions of part b so that you can call CART. or PØLAR. at any time (whether a number is in polar or cartesian). The procedure will put it in the proper form, possibly doing nothing. Modify your definitions of part a to enable them to accept numbers in either form, convert to the form the operation is performed in (polar for multiplication and division, and cartesian for addition) and return the answer in the form of the calculation. Also write a main program to calculate and print out the impedance of N impedances in parallel ($N \leq 50$) where the impedances are given in either polar or cartesian form (or mixed) and the results are to be printed out in both polar cartesian forms. Note that the impedance, z, of n impedances $z_1, z_2, \ldots, z_n$, in parallel is given by

$$z = \frac{1}{\frac{1}{z_1} + \frac{1}{z_2} + \ldots + \frac{1}{z_n}}$$

If any $z_i$ is zero, z is also zero.

# Chapter Eight

## ARRAYS, MATRICES AND LINEAR EQUATIONS

Many problems that are suited for machine solution can be most easily handled by using an array of variables having a common name with the individual elements being referred to by subscripts applied to that name. The dimensionality of an array is the number of subscripts that are used to reference it. One-dimensional arrays, often referred to as linear arrays, or vectors, were introduced in Chapter One. This chapter will be concerned primarily with the storage and application of two-dimensional arrays, although arrays of higher dimensionality will be discussed. An example of such an array, A, is

$$\begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{matrix}$$

Arrays are usually rectangular, i.e. they have the same number of elements in each row, column or other dimension.

### 8.1 Storage of Arrays

The memories of existing computers are addressed linearly, i.e. all locations are numbered in one consecutive sequence. In most of the programming languages currently in use, the elements of an array are stored in adjacent memory locations. Although other structures are possible, we will limit our discussion to this method of storage. Linear arrays fit naturally into this format. As was discussed in Chapter 3, in one common method of array storage (the one used in MAD) the elements of a linear array are stored in consecutive memory locations, with the highest subscript stored first. The base element of a linear array A, i.e. the one used for the calculation of the memory locations of all other elements of the array and the one listed in the memory map, is the element having a zero subscript. The memory location in which $a_i$ is stored is calculated by subtracting the value of the subscript, i, from the location of the base element. The instructions for this calculation are supplied automatically by the compiler whenever a subscript is used. All that the user need supply is the total number of memory locations that must be

reserved for the elements of the vector (using an appropriate declaration).

Another common arrangement is to store the base element (which may be the one with subscript 1) in the lowest position in memory and store the remaining elements in (consecutive) higher locations. Again, the user would be required to specify only the amount of memory required for the storage of the vector. The location of $a_i$ can then be calculated by adding i (i-1 if the base element has subscript 1) to the location of the base element.

In order to store, and later reference, arrays of dimensionality greater than one, one must specify their structure, as well as the amount of space required. Programming languages differ in the structures which are available for the storage of arrays; we discuss only the structure used in MAD. We first consider two-dimensional arrays and later generalize to arrays of higher dimensionality. In this structure, two-dimensional arrays are stored by row, i.e. consecutive elements in the same row of an array are stored in consecutive memory locations. Thus, the array A described above is stored in memory as though it were the vector:

$$a_{11} \quad a_{12} \quad a_{13} \quad a_{14} \quad a_{21} \quad a_{22} \quad \cdots \quad a_{33} \quad a_{34}$$

In order to calculate the memory location associated with each element of the array, a linear subscript corresponding to each set of multiple subscripts is first obtained. This linear subscript is then used to locate the element just as if it were a vector.

In order to establish the correspondence between linear and double subscripts, it is first necessary to specify the linear subscript corresponding to one double subscript. This element of the array is referred to as the base element. In MAD, the base element is the array element, all subscripts of which are one (e.g. $a_{1,1}$)[1]. In the three examples of Table 8.1, the linear subscripts of the base elements, b, are 1, 0, and 3, respectively. In addition to the linear subscript of the base element, the number of elements in each row is required in order to be able to calculate the linear subscript of any element not in the same row as the base element. Reference to Table 8.1 makes it clear that given the linear subscript, x, of some element $a_{i,j}$, the linear subscript, y, of the element in the same column and in the next row, $a_{i+1,j}$ is $y = x + 4$ or, in general, $y = x + n$, where n is the number of elements in each row. For example, in the last column, $a_{2,3}$ corresponds to $a_9$ and thus $a_{2+1,3}$ corresponds to $a_{9+4} = a_{13}$.

[1]The comma between subscripts is optional in mathematical notation, being used whenever there is a possibility of confusion.

Table 8.1 Examples of Array Storage

| Single Subscript (decimal) | Memory Location (octal) | Double Subscript | | |
|---|---|---|---|---|
| | | b = 1 | b = 0 | b = 3 |
| 0 | 10033 | none | 1 1 | none |
| 1 | 10032 | 1 1 | 1 2 | none |
| 2 | 10031 | 1 2 | 1 3 | none |
| 3 | 10030 | 1 3 | 1 4 | 1 1 |
| 4 | 10027 | 1 4 | 2 1 | 1 2 |
| 5 | 10026 | 2 1 | 2 2 | 1 3 |
| 6 | 10025 | 2 2 | 2 3 | 1 4 |
| 7 | 10024 | 2 3 | 2 4 | 2 1 |
| 8 | 10023 | 2 4 | 3 1 | 2 2 |
| 9 | 10022 | 3 1 | 3 2 | 2 3 |
| 10 | 10021 | 3 2 | 3 3 | 2 4 |
| 11 | 10020 | 3 3 | 3 4 | 3 1 |
| 12 | 10017 | 3 4 | none | 3 2 |
| 13 | 10016 | none | none | 3 3 |
| 14 | 10015 | none | none | 3 4 |

We can now develop a general formula to calculate the linear subscript, c, corresponding to an element $a_{i,j}$, given the linear subscript, b, of any other element, $a_{r,s}$.

$$c = b + (i-r)\cdot n + (j-s)$$

The linear subscript calculation contains three terms. The first term is the linear subscript of the base element, b. To this is added n for each row after the row containing the base element (or a multiple of n is subtracted if i is less than r). Thus, the first two terms locate the element in the same column as the base element, but in the i-th row. Finally, the last term in the sum, j-s, locates the element in the j-th column of the i-th row. As an example, if the array contains 4 columns, and the linear subscript of $a_{1,1}$ is chosen as 1 (as in the first example of

Table 8.1), then the formula to calculate the other elements of the array becomes

$$c = 1 + (i-1)\cdot 4 + j-1$$

Thus, for example, if i = 2 and j = 3, c is calculated to be 7, which agrees with the results in the table for b = 1.

The storage scheme can be extended to higher dimensional arrays in the following way. A three-dimensional array can be thought of as containing layers, rows and columns, where the first subscript indicates the layer, the second the row, and the third the column. The elements in each layer are stored in consecutive memory locations, as if each layer were a two-dimensional array. In order to calculate the linear subscript corresponding to an element in the same row and column of a known element but on the next layer, one need just add to the subscript of the known element, the number of elements in a layer. If there are n columns and m rows, there are $m\cdot n$ elements in a layer. Thus, the formula for the linear subscript, c, of the element $a_{h,i,j}$, given that the linear subscript of the element, $a_{q,r,s}$ is b, is

$$c = b + (h-q)\cdot m\cdot n + (i-r)\cdot n + (j-s)$$

The extension to arrays of higher dimensionality is obvious, and will not be discussed here.

## 8.2 Arrays in MAD

In this section, we will describe how arrays are used in MAD and the corresponding declarations that are required. Unlike vectors, the subscripts of arrays of dimensionality greater than one may have negative values, as well as positive and zero values. The only restriction is that the linear subscript corresponding to any element of an array must be non-negative. An element of an array may be referred to by its set of multiple subscripts or by the corresponding linear subscript.

In Chapter 3, the separate dimension statement was given, namely

DIMENSION L{ { <id> or ( L{ <id> } ) } ( <structure> )

where

<structure> is <uns-int-con> or <array-structure>

and the meta-variable <array-structure> was left to be defined here. Two forms of <array-structure> are available. We will first consider a simplified version that is adequate for most applications, and then present the more general form.

In the simplified form of array dimensioning, the user need only specify the maximum range of each of the subscripts. With this form of the dimensioning, the smallest linear subscript used is always 1, and thus the zero linear subscript is

not used. In the example of Table 8.1, this would produce the same storage assignment as shown in the column labeled b = 1.

When specifying the array structure, the subscript range is given by stating the lowest and highest values that subscript may take on or, if the lowest value is 1, by just stating the highest value. Thus,

Definition of subscript range:

<subscript-range> is <uns-int-con> or
( <uns-int-con>$_1$ ... <uns-int-con>$_2$ )

where

<uns-int-con>$_1$ $\leq$ <uns-int-con>$_2$

The range of each subscript is specified in this way, and thus, we have the

Definition of simplified array structure:

<array-structure-1> is L*{ <subscript-range> }

Two examples of the dimensioning of three-dimensional arrays are given in the following statement:

```
DIMENSIØN A(3*2*4), B((1...3)*2*(-1...2))
```

In the array A, the subscripts range from 1 through 3, 1 through 2, and 1 through 4, respectively, i.e. there are elements A(1,1,1), A(1,1,2), A(1,1,3), A(1,1,4), A(1,2,1), ..., A(3,2,4). The linear subscript 1 corresponds to the triple subscripts 1,1,1 and the linear subscript 24 corresponds to the triple subscripts 3,2,4. In the array B, the first two subscripts have the same range as those for A, but the third subscript ranges from -1 through 2. In this case B(1,1,-1) corresponds to B(1) and B(3,2,2) corresponds to B(24).

The span of a subscript is defined as the number of values that that subscript may have. In the case of the first form of <subscript-range>, the span just equals <uns-int-con>. In the second form, the span equals

<uns-int-con>$_2$ - <uns-int-con>$_1$ + 1

Thus, the span of the subscripts of both A and B in the above example are 3, 2, and 4, respectively.

If the structure of an array may vary depending on the data or calculations of the routine, the dimension declaration must indicate the maximum amount of memory to be used. If all subscripts are within the range specified in the declaration and if input and output block notations are not used, then the dimension information is sufficient for the operation of the routine.

Let us now consider what happens if a set of data contains a block of an array, for example

```
A(1,1) = 1, 2, 4, 1, -3, 6, 7, 2, 0 *
```

The first constant is stored in the location reserved for the specified element, A(1,1) in this case, and the succeeding constants are stored in consecutive memory locations. If these were to be the elements of an array with 3 rows and 3 columns, and the dimension statement indicated this, the array would be

```
1     2     4
1    -3     6
7     2     0
```

If the dimension statement had indicated a maximum size of five rows and five columns and nothing else had been done (e.g. DIMENSIØN A(5*5)), the array would be stored as

```
1     2     4     1    -3
6     7     2     0     -
-     -     -     -     -
-     -     -     -     -
-     -     -     -     -
```

where the dashes (-) indicate elements that were not filled by this data set. Their value would be the core constant unless they had been set earlier. Similarly, on output, the statement

```
PRINT RESULTS A(1,1) ... A(3,3)
```

will print all elements of the array from A(1,1) through A(3,3) in a block. If the array A had been set at 3 columns, the desired output would be produced, namely

```
A(1,1)...A(3,3)
   1     2     4
   1    -3     6
   7     2     0
```

However, if the structure indicated that the array contained five columns and the 3 by 3 array contained the desired values shown above (which would not be the case if the input data above were used), two columns in addition to those shown would be printed in the first two rows. These would contain the values of A(1,4), A(1,5), A(2,4), and A(2,5), which might well be core constant or remnants of some previous array.

In order to handle arrays that are smaller than the maximum size or simply structured differently, the subroutine SETDIM. may be used.

SETDIM.( <var-name>, L{ <subscript-change> } )

where <subscript-change> is <int-exp> or <int-exp> ... <int-exp>

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | SETDIM.(A,N,I...J) for two dimensional array A |
| Action: | The range of the first subscript of A is set to 1 through N and that of the second is set to I through J. SETDIM. does not change the amount of storage reserved for A. |

If the call

SETDIM.(A,3,3)

were made prior to the input or output statements discussed above, the desired results would be obtained as long as sufficient space had been reserved in the dimension statement.

The more versatile form used in the declaration of array structure is now given and the definition of array structure completed.

Definition of versatile array structure and array structure

<array-structure-2> is <uns-int-con> , <structure-vector>

<array-structure> is <array-structure-1> or <array-structure-2>

The constant in this form of the structure specification is one less than the number of memory locations to be reserved and thus is the largest linear subscript that may be used or calculated from multiple subscripts. The structure vector contains the remaining information about the structure of the array and is automatically declared to be in integer mode. Formally, the

Definition of structure vector:

<structure-vector> is <vector-base>

The elements of the structure vector contain, in order.

1) The dimensionality of the array.
2) The linear subscript corresponding to the base element, which has all subscripts 1. (e.g. A(1,1,1))
3) The span of the second subscript. If the dimensionality is greater than two, this vector must include additional elements for the span of the third and each succeeding subscript.

As is the case for any array, space must be reserved for the structure vector at compilation time. On the other hand, the values describing the structure of the array may be left unspecified until just before they are needed during execution. Both space reservation and assignment of values are often accomplished by presetting the

structure vector using a VECTØR VALUES statement, especially where some of the values are known at compilation time.

In order to illustrate these concepts, we will now consider a number of examples. A two-dimensional array with elements A(1,1) through A(6,5) (i.e. 6 rows and 5 columns, span of the second subscript 5) might be described either using

```
DIMENSIØN A(6*5)
```

or using

```
DIMENSIØN A(30,SVA)
VECTØR VALUES SVA = 2,1,5
```

In either case, space is reserved for linear subscripts up to 30, the linear subscript 1 corresponds to the double subscript 1,1 and A(6,5) corresponds to A(30). The zero linear subscript is not used, and thus that location in memory is wasted. Although any such correspondence between linear and multiple subscripts is allowed so long as all linear subscripts are non-negative, the smallest linear subscript is usually made zero or one to conserve storage space.

Consider next the two-dimensional array with elements running from Y(-4,0) through Y(10,4). One form of the dimension declaration would be

```
DIMENSIØN Y((-4...10)*(0...4))
```

where Y(-4,0) corresponds to Y(1) and Y(10,4) corresponds to Y(75). The span of the first subscript is 15 and that of the second subscript is 5. Thus, there are 75 elements in the array. Let us now consider the statements necessary to obtain the same storage allocation using the general form of the dimension declaration. Since the linear subscript corresponding to Y(-4,0) must be 1, we can use the formula

$$c = b + (i-r)\cdot n + (j-s)$$

with b = 1, r = -4, s = 0, i = 1, and j = 1 to calculate the linear subscript of the base element. The formula produces c = 27 and thus the declarations

```
DIMENSIØN Y(75,Q(5))
VECTØR VALUES Q(5) = 2,27,5
```

could be used. Note that the structural information is stored in Q(5), Q(6) and Q(7).

A four-dimensional integer mode array, with elements extending from P(1,1,1,1) through P(6,7,8,9) could be described using the declaration

```
INTEGER P(6*7*8*9)
```

or

```
INTEGER P(3023,D)
VECTØR VALUES D = 4,0,7,8,9.
```

Note that in the first case P(1,1,1,1) corresponds to P(1) and P(6,7,8,9)

corresponds to P(3024), whereas in the second case P(1,1,1,1) corresponds to P(0) and P(6,7,8,9) corresponds to P(3023).

In each of the preceding examples, the structure vector was set during the compilation of the routine. This is satisfactory if the same array structure is required throughout the execution of the routine. In many cases, however, a routine is designed to handle arrays of various sizes, e.g. an array A, with the first element A(1,1) and m rows and n columns. The largest such array may have 20 rows and 30 columns. The data includes the values of m and n, as well as the values of the elements of the array. A segment of the appropriate routine might be

```
DIMENSIØN A(20*30)
READ AND PRINT DATA   M,N,A(1,1)...A(M,N)
SETDIM.(A,M,N)
```

or

```
DIMENSIØN A(600,R)
VECTØR VALUES R = 2,1,0
READ AND PRINT DATA M,N,A(1,1)...A(M,N)
R(2) = N
```

In each case, the identical storage assignment is made, i.e. A(I,J) is stored in the same memory location. A typical set of data that might be used with this routine is

M = 2, N = 3, A(1,1) = 2.4, -6.9, 9.1, 0., -16.3, 4.1, 2.6*

Note that the structure of the array is not established until after the data is read. Double subscripts may be used in the data set only if the structural information provided by the data (the number of columns in A) is not needed. In general, this means that the only multiple subscripts that may be used are all 1's (except possibly for the last subscript).

## Exercises

1. For each of the arrays described, write a DIMENSIØN statement using the minimum amount of memory and a corresponding VECTØR VALUES statement to set the structure vector. Whenever possible also write a simplified DIMENSIØN statement.

   a) 17 columns and 12 rows from A(1,1) through A(12,17).

   b) 17 columns and 12 rows from A(-1,6) through A(10,22).

   c) 1 column and 1 row starting with A(-2,-5).

   d) 4 columns and 6 rows with A(1,1) corresponding to A(100) and the highest linear subscript equal to 120.

   e) a three-dimensional array with the span of the first subscript 4, the second 3, and the third 2, varying from A(0,0,0) through A(3,2,1) and A(0,0,0) corresponding to A(1).

2. A two dimensional array A is defined as follows:

```
DIMENSIØN A(100,B)
VECTØR VALUES B = 2,1,4
```

Also, A(0) is stored in location 11367 (octal).

a) In what location is A(5)?

b) What is the linear subscript corresponding to A(12,3)?

c) What is the double subscript corresponding to A(100)?

3. Repeat problem 2 for the array defined using

```
DIMENSIØN A((-1...18)*5)
```

4. An array A in a MAD program is described by the statements:

```
INTEGER A(25,D)
VECTØR VALUES D = 2,5,4
```

a) If the A(1,1) element is located in the machine at the octal address $20000_8$, what is the octal address of A(4,3)?

b) What value would SUM be assigned by the statement

```
SUM = A(8) + A(13) + A(4,3)
```

if just before obeying that statement the array contained

| | | | | |
|---|---|---|---|---|
| A(1,1) | (2) | 4 | 6 | 7 |
| | 8 | 9 | 3 | 1 |
| | 3 | 0 | 5 | 4 |
| | 6 | 5 | 8 | 0 |

## 8.3 Matrix Operations

In mathematics, two-dimensional rectangular arrays of numbers, on which certain operations are defined, are called matrices. Some of the basic operations on matrices are: multiplication of a matrix by a constant, addition of two matrices, and multiplication of two matrices[2]. In this section, we will develop an algorithm which tests two matrices for equality, as well as developing algorithms for these basic matrix operations.

Matrices are most commonly represented by upper case letters either in italics (e.g. $A$)[3] or in boldface (e.g. $\mathbf{A}$)[3]. When it is desired to show individual elements of a matrix, the most common notation uses a two-dimensional array enclosed in square brackets, e.g.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

A matrix having m rows and n columns is called a matrix of order (m,n) or an m × n matrix, read as "m by n".

The matrix shown above is of order (2,3). A square matrix is one having the same number of rows and columns (i.e. m = n). The upper left-hand element of a matrix usually has subscript 1,1 and, although this choice is arbitrary, we will restrict our discussion to that case. The major diagonal of a square matrix of order (n,n) consists of the elements $a_{11}$, $a_{22}$, ..., $a_{nn}$. The identity matrix, $I$, is a square matrix with all elements on the major diagonal equal to one and all other elements equal to zero.

The first algorithm to be presented is one to test whether two matrices are equal. Two matrices are defined as equal if and only if the matrices are of the same order and the corresponding elements of each matrix are equal. Thus, if $A$ is of order (m,n) and $B$ is of order (p,q), $A$ equals $B$ if and only if

(1) $m = p$,

(2) $n = q$,

(3) $a_{ij} = b_{ij}$ for all $i = 1,2,\ldots,m$ and $j = 1,2,\ldots,n$.

The algorithm will be designed in the form of a subroutine (a function) that returns

---

[2] It will be assumed in this section that the basic rules of matrix algebra are known to the reader. A good reference on this subject is F. E. Hohn, "Elementary Matrix Algebra," Second Edition, Macmillan, 1964.

[3] In typed manuscripts, italics are denoted by an underline and boldface by a tilda ( ~ ) underneath.

the value true (1B) if the matrices are equal and false (0B) otherwise. A flowchart for this subroutine named MATEQ., is shown in Figure 8.1a and the corresponding MAD coding in Figure 8.1b. Note that this function returns a Boolean value and thus, in the calling program, a declaration of the form

BØØLEAN MATEQ.

is required.

This algorithm, like most algorithms involving two-dimensional arrays, involves a pair of nested loops, the first or outer one controlling the row and the second the column. Two other algorithms with a similar loop structure are those for the multiplication of a matrix by a constant and the addition of two matrices. They each produce a matrix as a result and thus will be implemented as procedures.

To multiply a matrix by a constant, each element of the matrix is multiplied by that constant. Thus, if $\underline{A}$ is a matrix of order (m,n) and c is a constant, then

$$\underline{B} = c\underline{A}$$

is a matrix whose elements are given by

$$b_{ij} = c \cdot a_{ij}$$

for

$$i=1,2,\ldots,m \text{ and } j=1,2,\ldots,n.$$

This can be written as a procedure with dummy name list (A,m,n,c,B). A flowchart for this subroutine, named MATCØN., is shown in Figure 8.2.

Two matrices are conformable for addition, i.e. the operation of addition is defined for these matrices if and only if the matrices are of the same order. Thus, if $\underline{D}$ is of order (m,n) and $\underline{E}$ is of order (p,q), the sum $\underline{H} = \underline{D} + \underline{E}$ is defined if m = p and n = q. In that case, the elements of $\underline{H}$ are defined as

$$h_{ij} = d_{ij} + e_{ij}$$

for all i = 1,2,...,m and all j = 1,2,...n. The flowchart of this subroutine is similar to those for MATEQ. and MATCØN. and thus will not be shown. The MAD coding for this subroutine is shown in Figure 8.3.

Two matrices, $\underline{A}$ of order (m,n) and $\underline{B}$ of order (p,q), are conformable for multiplication in the order $\underline{A}\,\underline{B}$ (i.e. the product matrix $\underline{C} = \underline{A}\,\underline{B}$ exists) if and only if n = p. Matrix multiplication is not commutative, i.e. $\underline{A}\,\underline{B}$ is not, in general, equal to $\underline{B}\,\underline{A}$. In fact, the product $\underline{B}\,\underline{A}$ only exists if m equals q. Thus $\underline{A}\,\underline{B} = \underline{B}\,\underline{A}$ implies that m=n=p=q (although that condition is not sufficient). The terms of the product matrix $\underline{C}$ are given by:

$$c_{rs} = \sum_{i=1}^{n} a_{ri}\, b_{is}$$

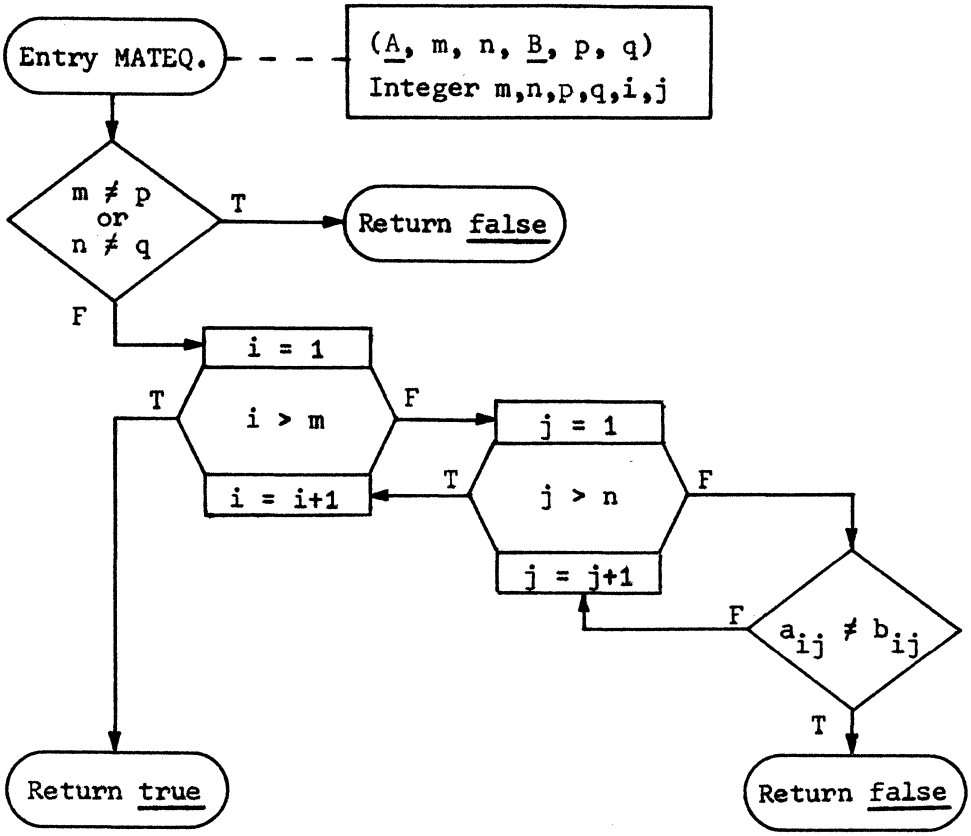



Figure 8.1a

```
      EXTERNAL FUNCTIØN (A,M,N,B,P,Q)
     RFUNCTIØN TØ TEST WHETHER THE MATRIX A ØF ØRDER (M,N) EQUALS
     R    THE MATRIX B ØF ØRDER (P,Q).  THE VALUE ØF THIS FUNCTIØN
     R    IS 1B IF THE MATRICES ARE EQUAL AND OB ØTHERWISE.
      INTEGER M,N,P,Q,I,J
      ENTRY TØ MATEQ.
      WHENEVER M.NE.P .ØR. N.NE.Q, FUNCTIØN RETURN OB
      THRØUGH S1, FØR I = 1,1,I.G.M
          THRØUGH S2, FØR J = 1,1,J.G.N
              WHENEVER A(I,J) .NE. B(I,J), FUNCTIØN RETURN OB
S2        CØNTINUE
S1        CØNTINUE
      FUNCTIØN RETURN 1B
      END OF FUNCTIØN
```

Figure 8.1b

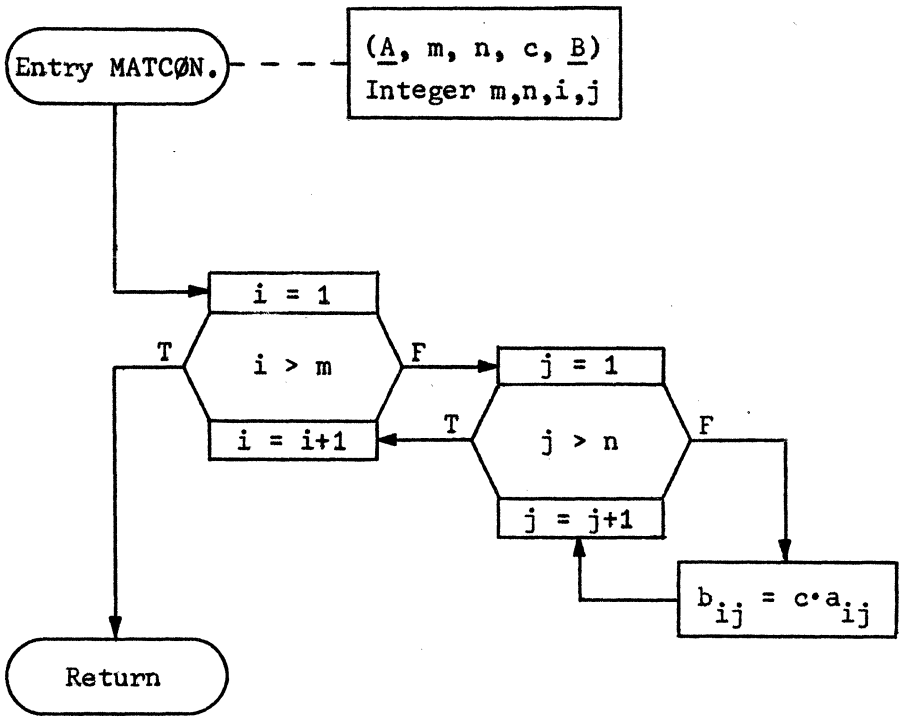



Figure 8.2 Flowchart for MATCØN.

```
      EXTERNAL FUNCTIØN (D,M,N,E,P,Q,H)
     RPRØCEDURE TØ ADD A MATRIX D ØF ØRDER (M,N) TØ A MATRIX E ØF
     R    ØRDER (P,Q).  THE SUM IS H.  AN ERRØR RETURN IS EXECUTED
     R    IF THE MATRICES ARE NOT CØNFØRMABLE.
      INTEGER M,N,P,Q,I,J
      ENTRY TØ MATADD.
      WHENEVER M.NE.P. ØR. N.NE.Q, ERRØR RETURN
      THRØUGH S1, FØR I = 1,1,I.G.M
          THRØUGH S2, FOR J = 1,1,J.G.N
              H(I,J) = D(I,J) + E(I,J)
S2        CØNTINUE
S1        CØNTINUE
      FUNCTIØN RETURN
      END ØF FUNCTIØN
```

Figure 8.3 Program for MATADD.

for all r=1,2,...,m and s=1,2,...,q. Thus $\underline{C}$ is of order (m,q).

As in the previous algorithms, two loops are involved in obtaining all the elements of $\underline{C}$ and the computation for each element involves a third (inner) loop. The flowchart for this subroutine, called MATMUL., is shown in Figure 8.4. The variable x is introduced in the calculation of $c_{rs}$ to reduce the number of repeated references to the same doubly subscripted element. Since each reference requires some computation, the use of x makes the program for this algorithm more efficient. It should be noted that this routine requires that the output matrix, $\underline{C}$, be distinct

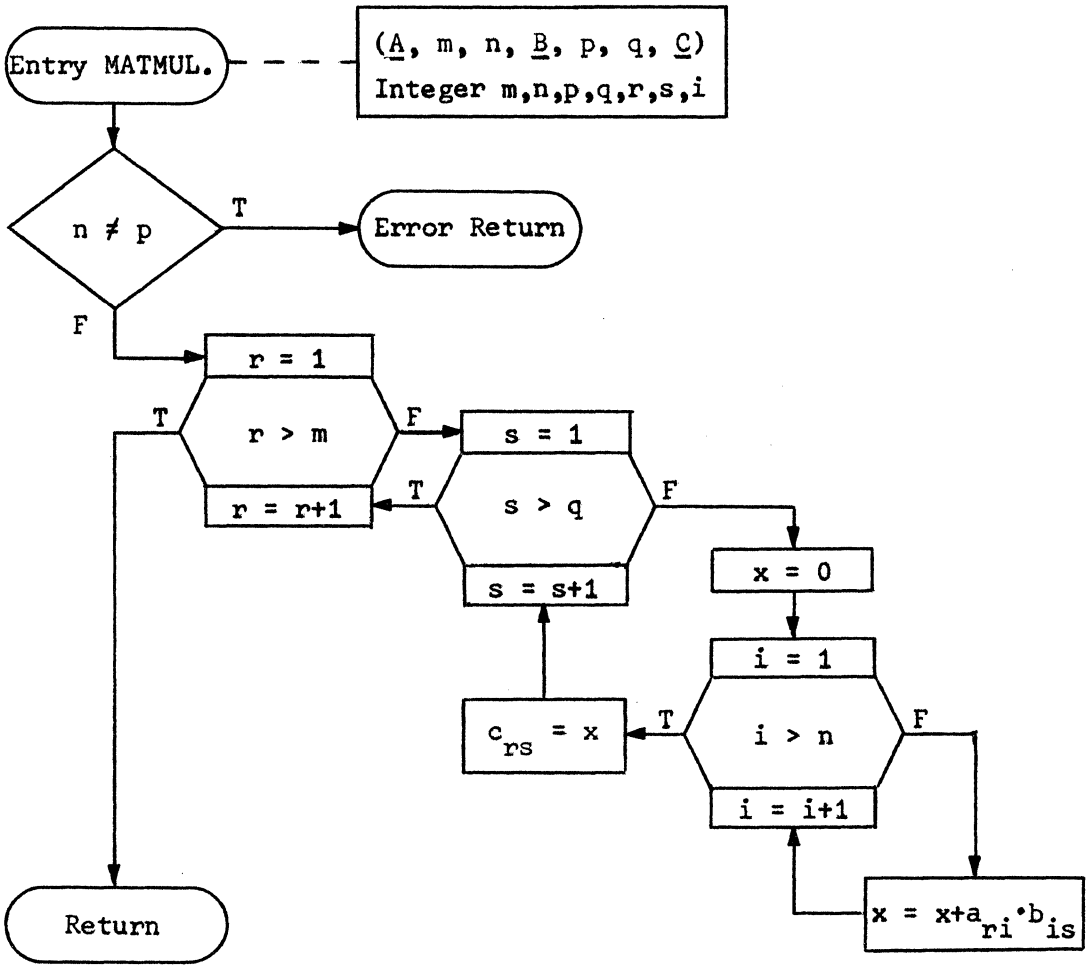



Figure 8.4 Flowchart for MATMUL.

from the input matrices, since elements of C are calculated when the corresponding elements of A and B are still needed. However, the order of C is not generally the same as that of A or B and thus this feature would not usually be very helpful. (See Exercise 5.)

## Exercises

1. Write a MAD program for the subroutine MATCØN. described in this section.

2. Draw a flowchart for the subroutine MATADD. described in this section.

3. Write a MAD program for the subroutine MATMUL. described in this section.

4. Draw a flowchart and write a MAD main program to calculate (making use of the subroutines described in this section)

$$\underline{B} = c_0\underline{I} + \sum_{i=1}^{k} c_i * \underline{A}^i$$

where A is a square matrix of order (n,n) and $A^i$ is defined as A*A*...*A (i product terms).

5. Draw a flowchart and write a MAD program for the procedure SQMMUL., with a typical dummy name list (A,B,n,C), where A and B are square matrices of order (n,n) and C = A B. This subroutine should be designed to allow calls of the form

```
SQMMUL. (A,B,N,A)  and
SQMMUL. (A,A,N,A).
```

6. a) The transpose of matrix is defined as follows: If a typical element of A is $a_{ij}$, then an element of the transpose, B is given by

$$b_{ij} = a_{ji}$$

Thus, if A is of order (m,n), B is of order (n,m). Draw a flowchart and write a MAD program for a procedure MATTR. with a typical dummy name list (A,m,n,B) that calculates $\underline{B} = \underline{A}^T$

b) Modify the routine of part a), calling it MATTR2., so as to allow calls of the form MATTR2. (A,M,M,A) if A is square.

## 8.4 The Gauss-Jordan Reduction - A Method of Solving Simultaneous Linear Equations

This section will be devoted to the theory of solving a set of m simultaneous linear equations in n unknowns, such as

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= a_{1,n+1} \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= a_{2,n+1} \\
&\cdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= a_{m,n+1}
\end{aligned}$$

We will show the Gauss-Jordan reduction, one of the most straightforward of the many procedures that will yield the solution of sets of equations in this form. Although algorithms exist for other methods (e.g. the Gauss elimination) that are more efficient computationally than those for Gauss-Jordan, we are presenting this one because of its clarity.

An attempted solution to a set of equations may yield one of three possible outcomes:

1) A unique solution exists, i.e. there exists exactly one set of x's $(x_1, x_2, \ldots, x_n)$ that satisfies the set of equations.
2) The equations are inconsistent, i.e. there exists no set of x's that satisfies the set of equations.
3) Many solutions exist, i.e. there exist many sets (always an infinite number of sets) of x's that satisfy the equations. Examples of each are

1) $x_1 + x_2 = 3$

$x_1 - x_2 = 1$

where $x_1 = 2$ and $x_2 = 1$ is the only solution.

2) $x_1 + x_2 = 3$

$2x_1 + 2x_2 = 4$

where no solution exists since $x_1 + x_2$ cannot equal 3 and 2 simultaneously.

3) $x_1 + x_2 = 3$

$2x_1 + 2x_2 = 6$

where $x_1 = 2$ and $x_2 = 1$; $x_1 = 0$ and $x_2 = 3$: and $x_1 = 63.42$ and $x_2 = -60.42$ are three of the infinite number of solutions.

We will consider the algorithm only for the case of a unique solution. Further, we will limit ourselves to the same number of equations as unknowns. We will use matrix notation, referring to the coefficient matrix

$$\underline{A}' = \begin{bmatrix} a_{11} & a_{12} \cdots a_{1n} \\ a_{21} & a_{22} \cdots a_{2n} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & a_{nn} \end{bmatrix}$$

and the augmented matrix,

$$\underline{A} = \begin{bmatrix} a_{11} & a_{12} \cdots a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} \cdots a_{2n} & a_{2,n+1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} \cdots a_{nn} & a_{n,n+1} \end{bmatrix}$$

which is just the coefficient matrix, plus an additional column containing the right-hand side of the equations. A reference to the i-th row of $\underline{A}$ is equivalent to a reference to the i-th equation in the set of equations and a reference to the j-th column of $\underline{A}$ is equivalent to a reference to the j-th unknown in the set of equations.

The Gauss-Jordan reduction for n equations in n unknowns involves n sets of transformations on the equations, each set often being referred to as a pivot. Each pivot results in eliminating one of the unknowns from all but one equation. The object of the n pivots is to end with each equation involving only one unknown, and thus have the solution, such as

$$\begin{aligned} x_1 &= b_1 \\ x_2 &= b_2 \\ &\cdots \\ x_n &= b_n \end{aligned}$$

Three transformations, often referred to as the elementary transformations, are used to reduce the equations. The elementary transformations have the property that they do not change the solution of the equations, i.e. any solution to the original equations is also a solution to the transformed equations, and vice versa. The three elementary transformations on a set of equations and the equivalent matrix row

transformations are:

1) interchanging any two equations (rows),
2) multiplying an equation (row) by a non-zero constant, and
3) adding a multiple of an equation (row) to any other equation (row).

Each pivot proceeds as follows[4]:

1) Choose an element of the coefficient matrix as the pivot element. It must be non-zero and must not be in a row or column of any previous pivot element. Call the pivot $a_{pq}$.
2) Multiply the pivot row of the augmented matrix (the p-th equation) by $\frac{1}{a_{pq}}$ (which sets the pivot element to unity).
3) For all $i \neq p$, add to the i-th row of the augmented matrix $-a_{iq}$ times the pivot row, thus setting each element in the pivot column (except the pivot element itself) to zero.

If n pivots can be performed, the equations will be transformed into the form of a set of equations each involving only a single unknown (as shown above).

One method of selecting pivots that automatically keeps track of which rows and columns have been used as pivots is to proceed down the major diagonal of the coefficient matrix, i.e. first use $a_{11}$, then $a_{22}$, and so forth. This procedure requires modification if $a_{ii}$ is zero at the time of the i-th pivot, since a zero pivot element is not allowed. Since the interchange of rows is allowed, a non-zero pivot element is obtained by interchanging the i-th row with the j-th row, a row that has not previously been used as the pivot. Using this method, j must be greater than i since pivots have already been chosen from the first i-1 rows. If $a_{ji}$ is zero for all $j \geq i$ (i.e. all elements in the pivot column on or below the major diagonal are zero), a unique solution does not exist. The modified algorithm is shown in Figure 8.5. Note that this is a flowchart for hand solution and not a complete flowchart for a computer program.

Example

$$\begin{aligned} 2x_1 + 3x_2 - 4x_3 &= 0 \\ 4x_1 + 6x_2 + x_3 &= 9 \\ x_1 + x_2 + 3x_3 &= 4 \end{aligned} \qquad \underline{A} = \begin{bmatrix} 2 & 3 & -4 & 0 \\ 4 & 6 & 1 & 9 \\ 1 & 1 & 3 & 4 \end{bmatrix}$$

[4]Note: The elements of the augmented matrix of both the original set of equations and each transformed set of equations are referred to as $a_{11}, a_{12}, \ldots a_{n,n+1}$. After each transformation the value of the elements of this matrix change; but each matrix is the augmented matrix of a set of equations that has the same solutions as the original set.

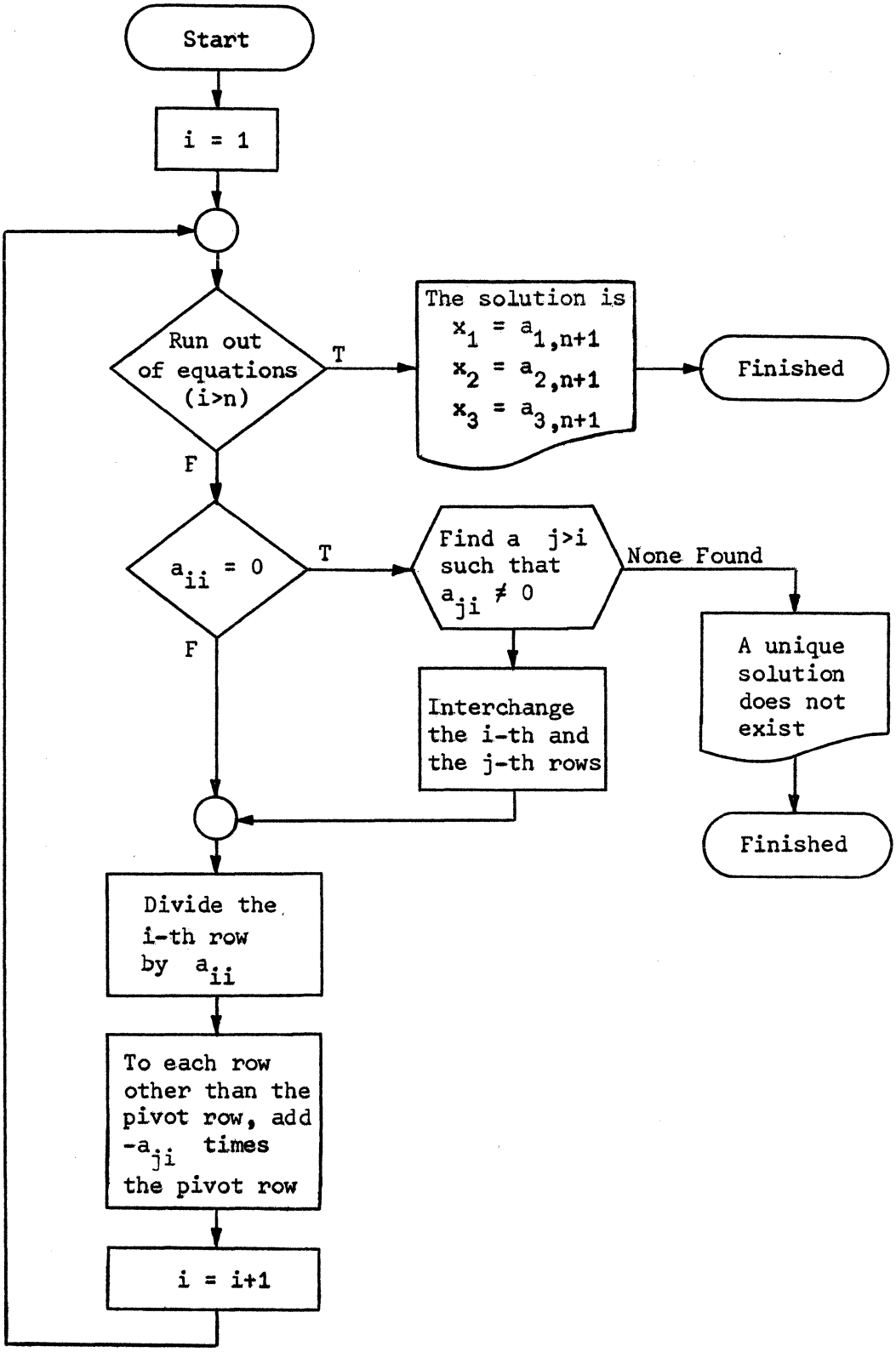

Start
i = 1
Run out of equations (i>n)
T
The solution is
$x_1 = a_{1,n+1}$
$x_2 = a_{2,n+1}$
$x_3 = a_{3,n+1}$
Finished
F
$a_{ii} = 0$
T
Find a j>i such that $a_{ji} \neq 0$
None Found
A unique solution does not exist
Finished
F
Interchange the i-th and the j-th rows
Divide the i-th row by $a_{ii}$
To each row other than the pivot row, add $-a_{ji}$ times the pivot row
i = i+1

Figure 8.5

First Pivot (using $a_{11} = 2$)

1) Multiply the first row by $\frac{1}{a_{11}} = \frac{1}{2}$, leaving

$$\underline{A} = \begin{bmatrix} 1 & 1.5 & -2 & 0 \\ 4 & 6 & 1 & 9 \\ 1 & 1 & 3 & 4 \end{bmatrix}$$

2) Add to the second row, $-a_{21}$ times the first row.

3) Add to the third row, $-a_{31}$ times the first row.

This leaves, at the end of the first pivot,

$$\underline{A} = \begin{bmatrix} 1 & 1.5 & -2 & 0 \\ 0 & 0 & 9 & 9 \\ 0 & -.5 & 5 & 4 \end{bmatrix}$$

Second Pivot

1) Since $a_{22}$ is zero, and $a_{32} \neq 0$, the second and third rows are interchanged.

2) The second row is then multiplied by (the new) $\frac{1}{a_{22}} = -2$, leaving

$$\underline{A} = \begin{bmatrix} 1 & 1.5 & -2 & 0 \\ 0 & 1 & -10 & -8 \\ 0 & 0 & 9 & 9 \end{bmatrix}$$

3) Add to the first row, $-a_{12}$ times the second row.

4) Add to the third row, $-a_{32}$ times the third row.

(Since $a_{32} = 0$, this step can be omitted in this example).

This leaves, at the end of the second pivot

$$\underline{A} = \begin{bmatrix} 1 & 0 & 13 & 12 \\ 0 & 1 & -10 & -8 \\ 0 & 0 & 9 & 9 \end{bmatrix}$$

Third Pivot (using $a_{33}$)

1) Multiply the third row by $\frac{1}{a_{33}}$ ( $= \frac{1}{9}$ )

2) Add to the first row, $-a_{13}$ times the third row.

3) Add to the second row, $-a_{23}$ times the second row.

This leaves, at the end of the third and last pivot

$$\underline{A} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Thus, the solution is

$$x_1 = -1$$
$$x_2 = 2$$
$$x_3 = 1$$

Exercises

Solve, by hand, the following sets of equations using the Gauss-Jordan elimination.

a) $x_1 + 2x_2 + x_3 = 4$

$x_1 - x_2 + x_3 = 5$

$2x_1 + 3x_2 - x_3 = 1$

b) $x_1 + 2x_3 = 7$

$2x_1 + x_3 = 5$

$4x_2 + 3x_4 = 20$

$3x_2 + 4x_4 = 18$

c) $8x_1 - 6x_2 + 8x_3 - 4x_4 = 1$

$-6x_1 + 9x_2 - 9x_3 + 4x_4 = 2$

$8x_1 - 9x_2 + 18x_3 - 2x_4 = -1$

$-4x_1 + 4x_2 - 2x_3 + 4x_4 = -2$

d) $x_1 + 2x_2 + 3x_3 = 3$

$4x_1 + 5x_2 + 6x_3 = 9$

$7x_1 + 8x_2 + 9x_3 = 15$

e) $x_1 + 2x_2 + 3x_3 = 4$

$4x_1 + 5x_2 + 6x_3 = 9$

$7x_1 + 8x_2 + 9x_3 = 15$

## 8.5 A Computer Algorithm for the Solution of Simultaneous Linear Equations.

In this section, we will present an algorithm for the computer solution of simultaneous linear equations. This algorithm will follow the one in the previous section for the manual solution of these equations. The first step in the development of this algorithm will be the presentation of three algorithms for the elementary transformations. We will then present an algorithm for a subroutine that executes a single pivot, making use of the three subroutines for the elementary operations. Finally, an algorithm for a main routine that calls the pivot subroutine will be presented. The MAD coding for these algorithms will be presented as a set of routines at the end of the section.

The algorithm presented in this section is being divided into five segments (four subroutines and one main routine). Segmentation is usually the best way to write complex programs, although the efficiency is somewhat less than if the entire algorithm were written as a single main program. If efficiency is important, the algorithm can be rewritten as a single program after the segmented program has been completely tested.

The first of the elementary transformations is the interchange of two rows of a matrix. Recall that in order to interchange two items, it is necessary to first store one of them in a temporary location. Thus if the values of a and b are to be interchanged, a set of statements such as

```
x = a
a = b
b = x
```

is necessary since

```
a = b
b = a
```

would result in the original value of b stored in both locations. A flowchart of a procedure, RØWINT., is shown in Figure 8.6. The j-th and k-th rows of the p column matrix A are interchanged.

The second elementary operation is the multiplication of a row by a constant. The algorithm is quite similar to that for row interchange, in that a single loop is involved in each. The dummy name list and the computational statements differ. A flowchart of an algorithm (entitled RØWMUL.) for the multiplication of the j-th row of a p column matrix A by x is shown in Figure 8.7. Note that a substitution is made for x so that no damage is done if the multiplier in the calling routine is an element of the j-th row of the matrix. (Note that this is not the case in our

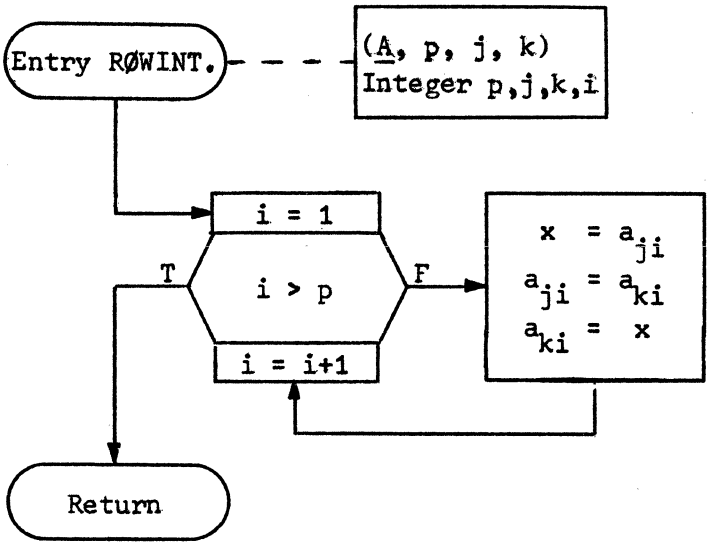



Figure 8.6 Flowchart of RØWINT.

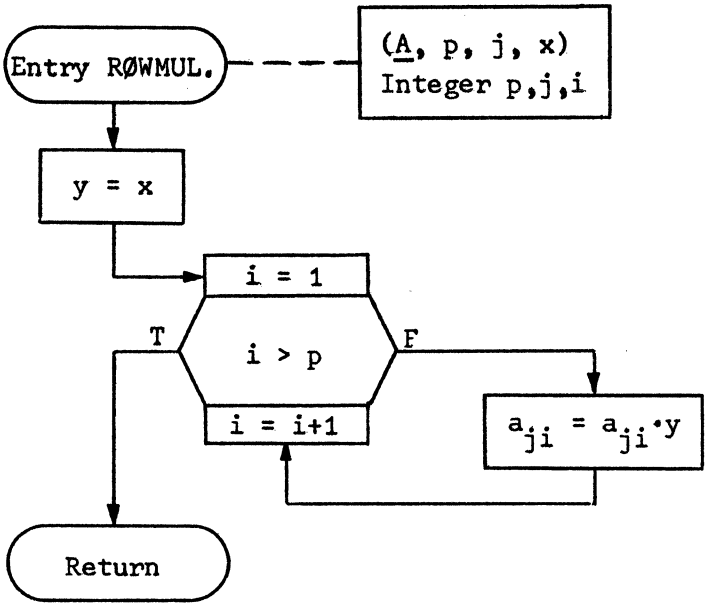



Figure 8.7 Flowchart of RØWMUL.

algorithm but would be if this subroutine were for the division of a row by a constant.)

The third elementary transformation, the addition of a constant multiple of one row to another row, can be obtained using a subroutine named ADDRØW.. The dummy name list is (A,P,j,k,x) where x times the j-th row is added to the k-th row. The flowchart for this procedure differs from that for RØWMUL. primarily in the computational statement (which becomes $a_{ki} = a_{ki} + x \cdot a_{ji}$) and will thus be omitted.

A flowchart of an algorithm (named PIVØT.) to perform a pivot on the augmented matrix of a set of n equations in n unknowns, using $a_{ii}$ as the pivot element is shown in Figure 8.8. The first portion of this algorithm obtains a non-zero element for $a_{ii}$. If this element is originally zero, a search of the remainder of that column is made in an attempt to find a non-zero element. If one is found, the appropriate rows of A are interchanged. Otherwise, an error return is executed, indicating that a unique solution does not exist. The second step in the pivot is to multiply the pivot row by the reciprocal of the pivot element. Finally, to each row other than the pivot row, the appropriate multiple of the pivot row is added so that the pivot column becomes zero everywhere except at the pivot itself.

The final routine in the algorithm for the machine solution of linear simultaneous equations is the main routine that calls PIVØT. n times and prints the result as a vector of x's. A flowchart of this algorithm is shown in Figure 8.9.

This program has several limitations. First, it can only handle the same number of equations as unknowns. Second, results are obtained only if a unique solution exists. Third, roundoff can produce serious errors if the set of equations is ill-conditioned[5]. Modifications of this algorithm are suggested in the exercises.

A listing of the program for this algorithm, including some typical sets of data, is given in Figure 8.10.

## Exercises

1. A single algorithm, using no subroutines, can be written to perform the Gauss-Jordan reduction. Also, the loops in RØWMUL. and ADDRØW. can be combined. Draw a flowchart and write a MAD main routine for this algorithm.

(Exercises continued on page 338)

[5] Fox, Leslie, Introduction to Numerical Linear Algebra, Clarendon Press, 1964, Oxford, Chapter 6.

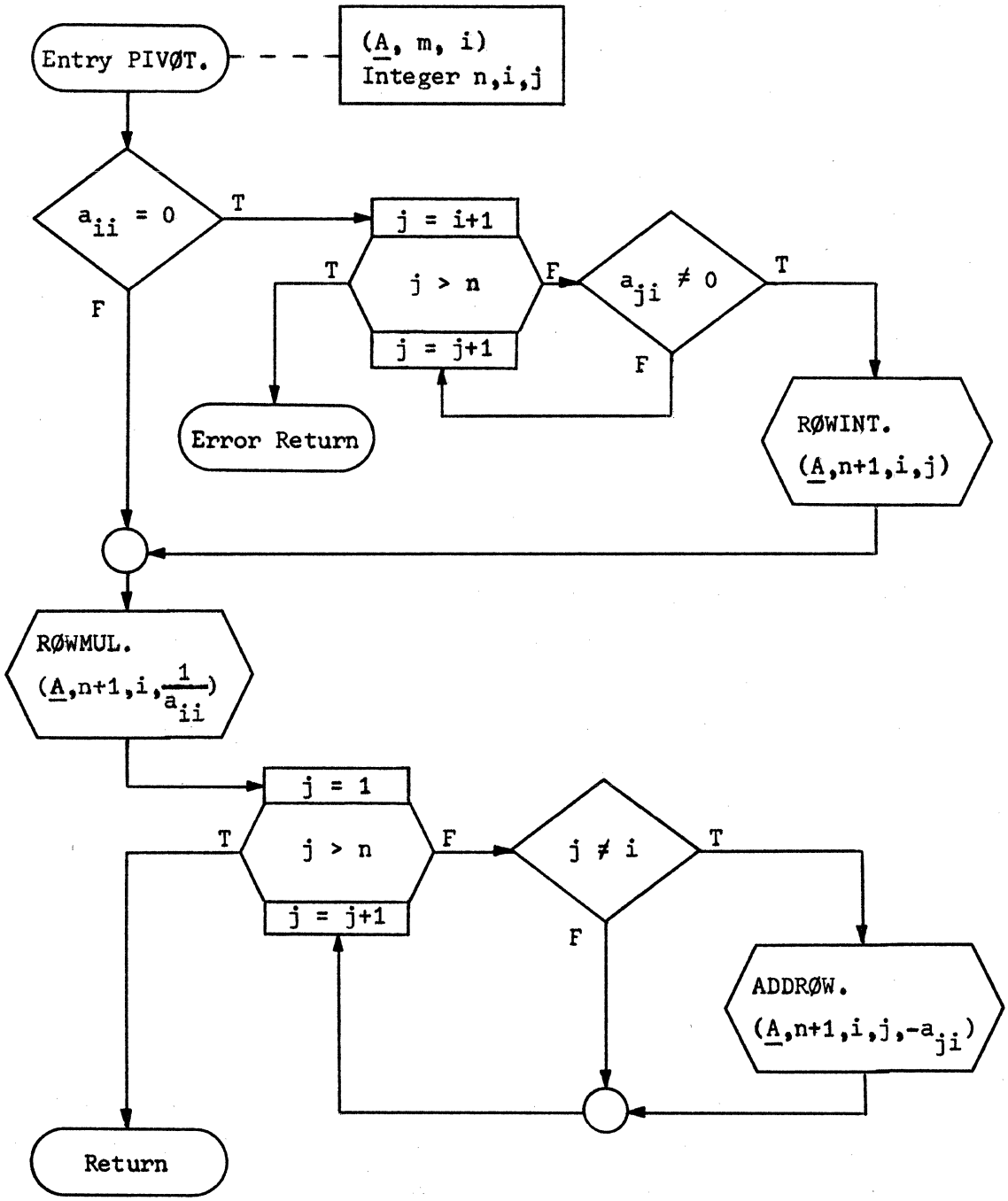



Figure 8.8 Flowchart of PIVØT.

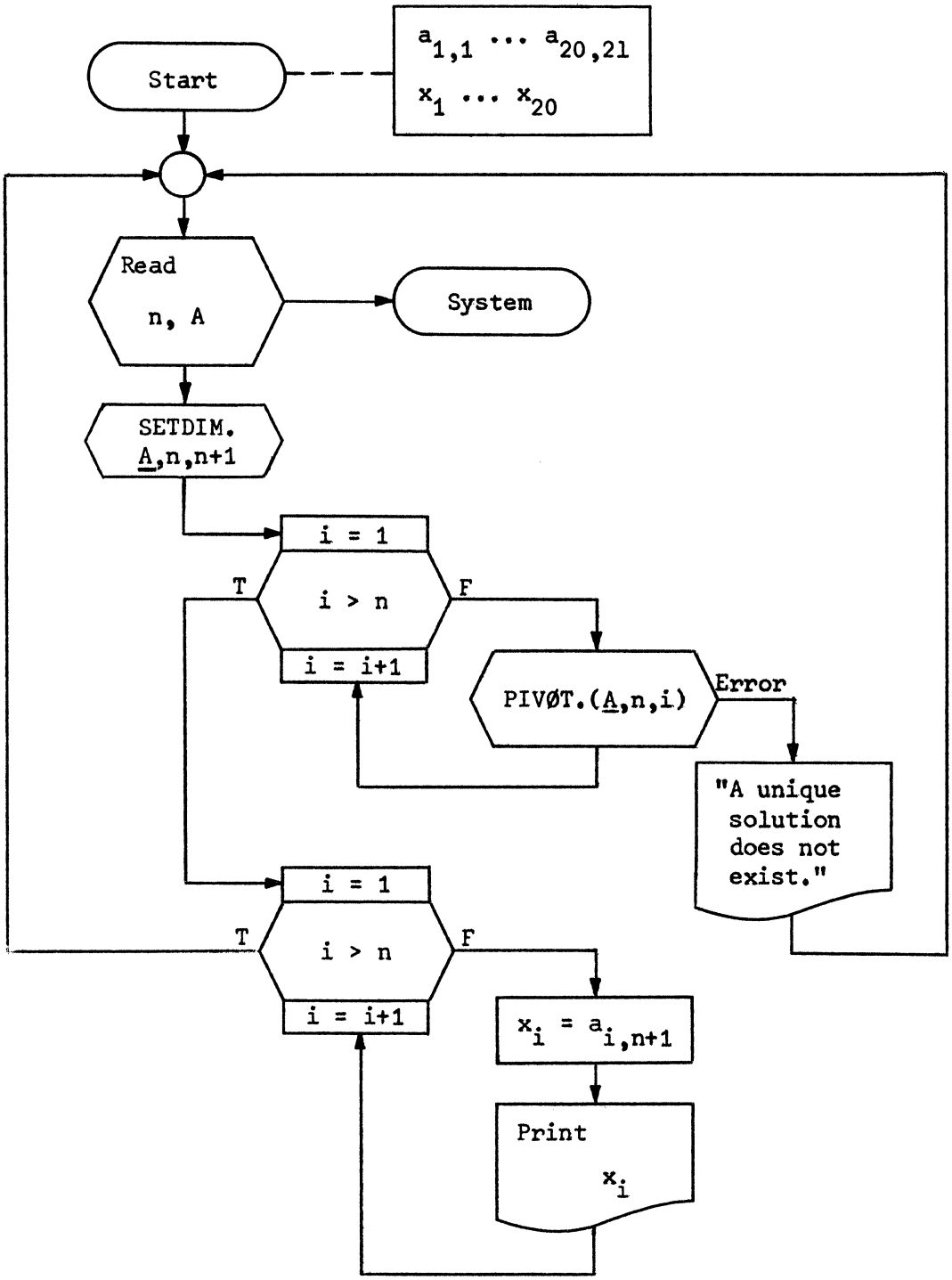



Figure 8.9 Flowchart of Main Program

```
$CØMPILE MAD, EXECUTE
          RSET ØF RØUTINES FØR THE SØLUTIØN ØF A SET ØF N SIMULTANEØUS
          R    LINEAR EQUATIØNS IN N UNKNØWNS.  THESE RØUTINES WILL
          R    PRØDUCE RESULTS ØNLY IF A UNIQUE SØLUTIØN EXISTS.
          R    NØ EFFØRT IS MADE TØ DIFFERENTIATE BETWEEN A SET ØF
          R    EQUATIØNS HAVING A NØN-UNIQUE SØLUTIØN AND ØNE HAVING
          R    NØ SØLUTIØN.
           INTEGER N,I
           DIMENSIØN A(20*21),X(20)
START      READ AND PRINT DATA          N,A(1,1)...A(N,N+1)
           SETDIM.(A,N,N+1)
           THRØUGH S, FØR I = 1,1,I.G.N
               PIVØT.(A,N,I,ERR)
S          CØNTINUE
           THRØUGH T, FØR I = 1,1,I.G.N
               X(I) = A(I,N+1)
               PRINT RESULTS X(I)
T          CØNTINUE
           TRANSFER TØ START
ERR        PRINT CØMMENT $0A UNIQUE SØLUTIØN DØES NØT EXIST.$
           TRANSFER TØ START
           END ØF PRØGRAM


$CØMPILE MAD
           EXTERNAL FUNCTIØN (A,N,I)
          RPRØCEDURE TØ EXECUTE A SINGLE PIVØT ABØUT THE A(I,I) ELEMENT.
          R    IT IS ASSUMED THAT PIVØTS HAVE BEEN EXECUTED USING A(J,J)
          R    FØR ALL J LESS THAN I.  THIS FUNCTIØN MØDIFIES A.
           INTEGER N,I,J
           ENTRY TØ PIVØT.
           WHENEVER A(I,I).E.0
               THRØUGH S1, FØR J = I+1,1,J.G.N
                    WHENEVER A(J,I).NE.0., TRANSFER TØ S2
S1             CØNTINUE
               ERRØR RETURN
S2             RØWINT.(A,N+1,I,J)
           END ØF CØNDITIØNAL
           RØWMUL.(A,N+1,I,1./A(I,I))
           THRØUGH S3, FOR J = 1,1,J.G.N
               WHENEVER J.NE.I, ADDRØW.(A,N+1,I,J,-A(J,I))
S3         CØNTINUE
           FUNCTIØN RETURN
           END ØF FUNCTIØN
```

Figure 8.10

```
$CØMPILE MAD
          EXTERNAL FUNCTIØN (A,P,J,K)
          RPRØCEDURE TØ INTERCHANGE THE J-TH RØW ØF THE K CØLUMN MATRIX
          R    A WITH THE K-TH RØW
          INTEGER P,J,K,I
          ENTRY TØ RØWINT.
          THRØUGH S1, FØR I = 1,1,I.G.P
              X = A(J,I)
              A(J,I) = A(K,I)
              A(K,I) = X
S1        CØNTINUE
          FUNCTIØN RETURN
          END ØF FUNCTIØN


$CØMPILE MAD
          EXTERNAL FUNCTIØN (A,P,J,X)
          RPRØCEDURE TØ MULTIPLY THE J-TH RØW ØF THE P CØLUMN MATRIX A
          R    BY X.
          INTEGER P,J,I
          ENTRY TO RØWMUL.
          Y = X
          THRØUGH S1, FØR I = 1,1,I.G.P
              A(J,I) = A(J,I)*Y
S1        CØNTINUE
          FUNCTIØN RETURN
          END ØF FUNCTIØN


$CØMPILE MAD
          EXTERNAL FUNCTIØN (A,P,J,K,X)
          RFUNCTIØN TØ ADD X TIMES THE J-TH RØW TØ THE K-TH RØW ØF A
          R    P CØLUMN MATRIX A.
          INTEGER P,J,K,I
          ENTRY TØ ADDRØW.
          Y = X
          THRØUGH S1, FØR I = 1,1,I.G.P
              A(K,I) = A(K,I) + Y*A(J,I)
S1        CØNTINUE
          FUNCTIØN RETURN
          END ØF FUNCTIØN


$DATA
N=3,A(1,1) = 1.,1.,1.,3.,1.,2.,3.,6.,1.,-2.,1.,0.*
N=2,A(1,1) = 1.,2.,3.,4.,5.,6.*
N=2,A(1,1) = 3.,1.,2.,6.,2.,3.*
```

Figure 8.10 (Continued)

2. Roundoff errors often pose a serious problem in the solution of simultaneous linear equations. One method of reducing these errors is to choose as the pivot element for the i-th pivot the largest element in magnitude on or below the major diagonal in the i-th column. Modify the flowchart and MAD subroutine for PIVØT. to choose the pivot in that way.

3. Draw a flowchart and write a MAD subroutine, GAUSJ., with dummy name list (A,n,X), that calculates, using the algorithm of this section a unique solution to the set of n equations in n unknowns, the augmented matrix of which is A. The solution is given in the vector X ($x_1 \ldots x_n$). The error condition is that a unique solution does not exist. This subroutine must not modify A.

4. Modify the main algorithm and PIVØT. (obtaining PIVØT1. with dummy name list (A,m,n,i)) so as to solve m equations in n unknowns where m may or may not equal n. Consider only the special case where it is possible to find n non-zero pivots. At the end of these n pivots, the last (m-n) equations are all of the form

$$0 = a_{j,n+1}$$

i.e. the last (m-n) rows of the coefficient matrix are all zero. If any one of the $a_{j,n+1}$ (for j=n+1,n+2,...,m) is non-zero, then no solution exists. Otherwise, the solution is $x_j = a_{j,n+1}$ for j = 1,2,...,n, as before.

5. Due to roundoff errors, the solution obtained using the Gauss-Jordan algorithm will not be exact, i.e., if $x_1,\ldots,x_n$ are substituted into the original equations, modified to the form

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n - a_{1,n+1} = 0$$

the value of the left hand side will not be zero, but some remainder $r_1$. If the remainders are used to replace the right hand side, i.e. if the augmented matrix becomes

$$\begin{matrix} a_{11} & a_{12} \ldots a_{1n} & r_1 \\ - & - \ - \ - \ - \ - \ - \ - & - \\ a_{n1} & a_{n2} \ldots a_{nn} & r_n \end{matrix}$$

and a solution, $y_1, y_2, \ldots, y_n$, to these equations is found, then

$$(x_1-y_1), (x_2-y_2), \ldots, (x_n-y_n)$$

is normally a better solution to the original set of equations. Draw a flowchart and write a MAD main routine to calculate this improved solution, using the subroutines defined in this section.

Bibliography

Arden, B. W. *An Introduction to Digital Computing.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1963.

_____, B. A. Galler, and R. M. Graham. *The Michigan Algorithm Decoder.* Ann Arbor, Mich.: University of Michigan Computing Center, 1965.

Backus, J. W. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference," *Proc. Internat. Conf. Inf. Proc.* Paris: UNESCO, June 1959.

Bernstein, Jeremy. *The Analytical Engine: Computers--Past, Present, and Future.* New York: Random House, 1963.

Brooks, F. P., and K. E. Iverson. *Automatic Data Processing.* New York: John Wiley & Sons, Inc., 1963.

Calingaert, Peter. *Principles of Computation.* Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1965.

Carnahan, Brice, H. A. Luther, and J. O. Wilkes. *Applied Numerical Methods* --Preliminary Edition. New York: John Wiley & Sons, Inc., 1964.

Conte, S. D. *Elementary Numerical Analysis, An Algorithmic Approach.* New York: McGraw-Hill Book Company, 1965.

Desmonde, W. H. *Computers and Their Uses.* Englewood Cliffs, N. J.: Prentice-Hall, 1964.

Feigenbaum, E. A., and Julian Feldman. (ed.) *Computers and Thought.* New York: McGraw-Hill Book Company, 1963.

Fox, Leslie. *Introduction to Numerical Linear Algebra.* Oxford: Clarendon Press, 1964.

Galler, B. A. *The Language of Computers.* New York: McGraw-Hill Book Company, 1962.

Gregory, R. H., and R. L. Van Horn. *Automatic Data-Processing Systems.* San Francisco: Wadsworth Publishing Company, Inc., 1960.

Greenberger, Martin. (ed.) *Computers and the World of the Future.* Cambridge, Mass: The MIT Press, 1962.

Hamming, R. W. *Numerical Methods for Scientists and Engineers.* New York: McGraw-Hill Book Company, 1962.

Henrici, Peter. *Elements of Numerical Analysis.* New York: John Wiley & Sons, Inc., 1964.

Hildebrand, F. B. *Introduction to Numerical Analysis.* New York: McGraw-Hill Book Company, 1956.

Hohn, F. E. *Applied Boolean Algebra, An Elementary Introduction.* New York: The Macmillan Company, 1960.

_____. *Elementary Matrix Algebra.* 2d ed. New York: The Macmillan Company, 1964.

Jennings, Walter. *First Course in Numerical Methods.* New York: The Macmillan Company, 1964.

Markov, A. A. *Theory of Algorithms,* translated from Russian. Washington, D. C.: Office of Technical Services, 1961.

Naur, Peter. (ed.) "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the* ACM, Vol. 6, No. 1 (Jan. 1963), 1-17.

Nielsen, K. L. *Methods in Numerical Analysis.* 2d ed. New York: The Macmillan Company, 1964.

Organick, E. I. *A MAD Primer.* Houston, Texas: University of Houston, 1964.

Pennington, R. H. *Introductory Computer Methods and Numerical Analysis.* New York: The Macmillan Company, 1964.

Richards, R. K. *Arithmetic Operations in Digital Computers.* Princeton, N. J.: D. Van Nostrand Company, Inc., 1955.

School Mathematics Study Group. *Algorithms, Computation and Mathematics.* Stanford, Calif.: Leland Stanford Junior University, 1965.

Sherman, P. M. *Programming and Coding Digital Computers.* New York: John Wiley & Sons, Inc., 1963.

Whitesitt, J. E. *Boolean Algebra and Its Applications.* Reading, Mass. Addison-Wesley Publishing Company, Inc., 1961.

Wilkinson, J. H. *Rounding Errors in Algebraic Processes.* Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1963.

# Appendix A

## ABBREVIATIONS OF KEY PHRASES IN MAD

| | |
|---|---|
| B'E | BACKSPACE FILE ØF TAPE (not BACKSPACE RECØRD ØF TAPE) |
| B'N | BØØLEAN |
| C'E | CØNTINUE |
| D'N | DIMENSIØN |
| E'L | END ØF CØNDITIØNAL |
| E'N | END ØF FUNCTIØN (not EXTERNAL FUNCTIØN or ERRØR RETURN) |
| E'M | END ØF PRØGRAM |
| E'Ø | ENTRY TØ |
| E'E | ERASABLE (not EXECUTE or EQUIVALENCE or END ØF FILE TAPE) |
| F'T | FLØATING PØINT |
| F'E | FØRMAT VARIABLE (not FUNCTIØN NAME) |
| F'F | FØR VALUES ØF |
| F'N | FUNCTIØN RETURN |
| I'Ø | IF LØAD PØINT TRANSFER TØ |
| I'R | INTEGER |
| I'N | INTERNAL FUNCTIØN |
| L'T | LØØK AT FØRMAT |
| M'R | MØDE NUMBER |
| N'S | NØRMAL MØDE IS |
| N'N | NØRMAL MØDE IS BØØLEAN |
| N'T | NØRMAL MØDE IS FLØATING PØINT |
| N'E | NØRMAL MØDE IS FUNCTIØN NAME |
| N'R | NØRMAL MØDE IS INTEGER |
| N'L | NØRMAL MØDE IS STATEMENT LABEL |
| Ø'R | ØR WHENEVER |
| Ø'E | ØTHERWISE |
| P'T | PRINT FØRMAT (not PRINT CØMMENT or PUNCH FØRMAT or PRINT ØN LINE FØRMAT) |
| P'S | PRINT RESULTS (not PRINT BCD RESULTS or PRINT ØCTAL RESULTS) |
| P'N | PRØGRAM CØMMØN |
| R'A | READ AND PRINT DATA (not READ DATA or RESTØRE DATA) |
| R'E | READ BINARY TAPE (not READ BCD TAPE OR REWIND TAPE) |
| R'T | READ FØRMAT |
| S'A | SAVE DATA |
| S'N | SAVE RETURN |
| S'Ø | SET LIST TØ |
| S'E | SET LØW DENSITY TAPE (not SET HIGH DENSITY TAPE) |
| S'L | STATEMENT LABEL |
| T'H | THRØUGH |
| T'Ø | TRANSFER TØ |
| U'E | UNLØAD TAPE |
| V'S | VECTØR VALUES |
| W'R | WHENEVER |
| W'E | WRITE BINARY TAPE (not WRITE BCD TAPE) |

Note that each of these abbreviations is formed by using the first and last letter of the phrase.

## Appendix B

## NUMBER SYSTEMS

Since even the most primitive man seems to have possessed a certain amount of "number sense", it is uncertain just when man first became aware of the concept of numbers. Archaelogical findings in caves in Europe, Africa, and Asia indicate that numeration -- the recording of numbers by written symbols -- is at least as old as written language. However, the earliest indications of the systematic use of written numerals to express numbers have been traced to the Egyptian and Sumerian civilizations of around 3500 B.C.

The Egyptian system, like those used later by the Hebrews, Greeks, and Romans, was based on a purely additive principle. The most simple such system requires only a mark, say "I", for "one" which is simply used as many times as necessary to represent the desired number. For example, "five" would be represented as IIIII. Since this leads to very long representations of numbers, additional symbols having greater "value" were introduced. For example, the Romans used V, X, L, C, D and M. Thus, CLXXXIII represented the number "one hundred eighty-three" in the Roman notation with its value being obtained by adding the values of the individual symbols as follows:

C + L + X + X + X + I + I + I

one hundred + fifty + ten + ten + ten + one + one + one

If we restrict ourselves to the purely additive system which was used initially by the Romans, then the number "four" must be represented as "IIII" instead of "IV". In such systems, the set of symbols representing a number may be written in any order without affecting the value of the number represented. Thus, "thirty-eight" could be represented by "XXXVIII", "XIIVXIX", "VIXXIIX", "IIIVXXX", or several other arrangements. One of the disadvantages of such systems of notation is that in order to avoid lengthy number representations more new symbols are needed as the numbers to be represented become larger. Another disadvantage is that operations, such as addition and multiplication on numbers represented in this way are very difficult to perform.

The Sumerian system, on the other hand, was one in which the numerical value of a symbol depended on the position of that symbol within the entire representation. For example, in the positional notation commonly used today, the symbol "3" in "183"

represents the number "three" while the same symbol represents "thirty" when written in "138" and "three hundred" when written in "381." The attributes possessed by this Sumerian invention, which was later used by the Babylonians and further developed by the Hindus, make it a significant development in the history of civilization.

In order to investigate the attributes of positional notation which make it of such importance, let us momentarily direct our attention to the so-called Hindu-Arabic notation used today. (Hindu-Arabic refers to the origin of the symbols used and how they were transmitted from India to Europe over a thousand years ago.) The representation of the number "five thousand two hundred seventy-three" by "5273" in this notation is really an abbreviated way of writing the polynomial

$$5 \times 10^3 + 2 \times 10^2 + 7 \times 10^1 + 3 \times 10^0$$

Writing the number as a polynomial enables us to understand immediately why the position of the digits is so important and why the symbol "3" represents different values in "183", "138", and "381". It also reveals one of the advantages of positional notation over the purely additive notation mentioned above: only the ten symbols, "0", "1", "2", "3", "4", "5", "6", "7", "8", and "9", are needed to represent any integer in the decimal number system, no matter how large. These ten symbols are commonly called the "digits" after the Latin digitus which means finger or toe. Another advantage of this positional notation is that computation is quite simple, because the rules for addition and multiplication can be easily learned after one has memorized the addition and multiplication tables for the digits themselves. Thus, computation in positional notation can now be taught in elementary school, whereas the complexities of addition, subtraction, multiplication, and division in purely additive notation systems, such as Roman numerals, prevented all but the most adept from learning how to perform these calculations in ancient times.

## B.1 Positional Notation for Integers

There seems to be no particular reason why the number ten has come to be used by man as the most common base for his number representation, other than the fact that he has ten fingers on his two hands. Indeed, bases other than ten have been used at various times in history; the Sumerians and Babylonians used a base of sixty, the Greeks used twenty for a time (presumably because they wore open-toed sandals), and hints of a base of five are discernible in the Roman numeral system and in the abacus. In theory, one can use any integer greater than one as a base

or radix since any non-negative integer can be represented as a polynomial in b,

$$\ldots\ d_n b^n + d_{n-1} b^{n-1} + \ldots + d_2 b^2 + d_1 b^1 + d_0 b^0$$

where b is an integer greater than 1. As in the case of our decimal system, it is sufficient and more convenient to omit the powers of b and all the zero coefficients preceding the first non-zero coefficient and to write only the coefficients, $d_n d_{n-1} \ldots d_2 d_1 d_0$, once the base has been agreed upon.

Although coefficients $d_n$, $d_{n-1}$, ..., $d_2$, $d_1$, and $d_0$ may be any integer, only b distinct digits representing 0, 1, ..., b-1, are actually necessary. Moreover, it can be shown that, when just these values are permitted for the coefficients, the representation of every integer is unique; that is, different representations must represent different numbers. Negative integers are represented in positional notation by prefixing the entire string of coefficients with a minus sign (-). For example, in the base five or "quinary" system, the decimal number "one hundred eighty-three" is represented by "1213" since

$$183_{10} = 1213_5 = 1\times5^3 + 2\times5^2 + 1\times5^1 + 3\times5^0$$

Similarly, the same number in the "binary" (base 2), "ternary" (base 3), "septimal" (base 7), and "duodecimal (base 12) systems is:

$$10110111_2 = 1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^4 + 1\times2^1 + 1\times2^0$$

$$20210_3 = 2\times3^4 + 0\times3^3 + 2\times3^2 + 1\times3^1 + 0\times3^0$$

$$351_7 = 3\times7^2 + 5\times7^1 + 1\times7^0$$

$$133_{12} = 1\times12^2 + 3\times12^1 + 3\times12^0$$

and the number "one hundred thirty-eight" in the "octal" (base 8), "decimal", (base 10) and "hexadecimal" (base 16) systems is:

$$212_8 = 2\times8^2 + 1\times8^1 + 2\times8^0$$

$$138_{10} = 1\times10^2 + 3\times10^1 + 8\times10^0$$

$$8A_{16} = 8\times16^1 + A\times16^0$$

Here we observe that for bases greater than ten we must invent some new digits. In the "hexadecimal" system the symbols A, B, C, D, E, F are usually used to represent the numbers ten, eleven, twelve, thirteen, fourteen and fifteen, respectively. In the "duodecimal" system the symbols T and E are often used for ten and eleven. The representations of zero and the first twenty positive integers in several common systems are shown in Table B.1. Note that, for the smaller bases, more non-zero coefficients from the polynomial must be written in order to express a given number.

Table B.1 Counting Numbers

| Decimal | Binary | Ternary | Quinary | Septimal | Octal | Duodecimal | Hexadecimal |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 11 | 10 | 3 | 3 | 3 | 3 | 3 |
| 4 | 100 | 11 | 4 | 4 | 4 | 4 | 4 |
| 5 | 101 | 12 | 10 | 5 | 5 | 5 | 5 |
| 6 | 110 | 20 | 11 | 6 | 6 | 6 | 6 |
| 7 | 111 | 21 | 12 | 10 | 7 | 7 | 7 |
| 8 | 1000 | 22 | 13 | 11 | 10 | 8 | 8 |
| 9 | 1001 | 100 | 14 | 12 | 11 | 9 | 9 |
| 10 | 1010 | 101 | 20 | 13 | 12 | T | A |
| 11 | 1011 | 102 | 21 | 14 | 13 | E | B |
| 12 | 1100 | 110 | 22 | 15 | 14 | 10 | C |
| 13 | 1101 | 111 | 23 | 16 | 15 | 11 | D |
| 14 | 1110 | 112 | 24 | 20 | 16 | 12 | E |
| 15 | 1111 | 120 | 30 | 21 | 17 | 13 | F |
| 16 | 10000 | 121 | 31 | 22 | 20 | 14 | 10 |
| 17 | 10001 | 122 | 32 | 23 | 21 | 15 | 11 |
| 18 | 10010 | 200 | 33 | 24 | 22 | 16 | 12 |
| 19 | 10011 | 201 | 34 | 25 | 23 | 17 | 13 |
| 20 | 10100 | 202 | 40 | 26 | 24 | 18 | 14 |

On the other hand, fewer symbols are needed for these smaller bases so that the addition and multiplication tables are correspondingly reduced.

Exercise

1. What decimal is represented by 1201 in base 8? Base 5? Base 3?

## B.2 Arithmetic Operations on Integers

As we mentioned before, computation in positional notation is quite easy given the addition and multiplication tables for the digits. Consider for example the problem of adding 213 to 324 in base five. Using Table B.2a, we see from the entry in the "4" row and "3" column that 4 + 3 is 12 in base five. Writing down the 2 and "carrying" the 1, we again use Table B.2a to add 1, 2, and (the carried) 1 to obtain 4. (Note that two separate additions are actually performed, but it makes no difference which two numbers are added first, since addition is commutative and associative regardless of the base.) Finally, we add the leftmost digits, 2 and 3, to obtain 10, so that in base five we have:

$$\begin{array}{r} {\scriptstyle 1\ \ } \\ 3\ 2\ 4_5 \\ +\ 2\ 1\ 3_5 \\ \hline 1\ 0\ 4\ 2_5 \end{array}$$

In base two or in any other base, addition is performed in exactly the same way. For example, we may use Table B.3a to add $10110_2$ to $10111_2$ as follows:

$$\begin{array}{r} {\scriptstyle 1\ 1\ \ \ \ } \\ 1\ 0\ 1\ 1\ 1_2 \\ +\ 1\ 0\ 1\ 1\ 0_2 \\ \hline 1\ 0\ 1\ 1\ 0\ 1_2 \end{array}$$

Some further examples of addition follow:

$$\begin{array}{r} {\scriptstyle 1\ \ } \\ 1\ 2\ 0\ 1_3 \\ +\ 2\ 0\ 1\ 2_3 \\ \hline 1\ 0\ 2\ 2\ 0_3 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\ 1\ \ \ } \\ 1\ 7\ 4\ 5_8 \\ +\ 4\ 3\ 6\ 2_8 \\ \hline 6\ 3\ 2\ 7_8 \end{array} \qquad \begin{array}{r} {\scriptstyle 1\ \ } \\ 1\ T\ 7_{12} \\ +\ E\ E\ 4_{12} \\ \hline 1\ 1\ 9\ E_{12} \end{array} \qquad \begin{array}{r} {\scriptstyle 1\ \ } \\ 1\ A\ 2\ E_{16} \\ +\ F\ 0\ 9\ C_{16} \\ \hline 1\ 0\ A\ C\ A_{16} \end{array}$$

Table B.2 Base 5 Operations

(a)

| + | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 |
| 1 | 1 | 2 | 3 | 4 | 10 |
| 2 | 2 | 3 | 4 | 10 | 11 |
| 3 | 3 | 4 | 10 | 11 | 12 |
| 4 | 4 | 10 | 11 | 12 | 13 |

(b)

| × | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 |
| 2 | 0 | 2 | 4 | 11 | 13 |
| 3 | 0 | 3 | 11 | 14 | 22 |
| 4 | 0 | 4 | 13 | 22 | 31 |

Table B.3 Base 2 Operations

(a)

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

(b)

| × | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

As in the familiar decimal arithmetic, subtraction can be performed by using the addition table and "working backwards." One must remember that a "borrow" of one from the next higher order column has as its value the base of the number system being used. Thus, when subtracting 6 from 4 in decimal, the borrow produces a subtraction of 6 from fourteen, whereas, in octal, the subtraction is 6 from twelve (four plus eight). Some examples of subtraction are:

$$\begin{array}{r} 1\,1\,0\,1\,1_2 \\ -\,1\,0\,1\,1\,0_2 \\ \hline 1\,0\,1_2 \end{array} \qquad \begin{array}{r} 4\,1\,0_5 \\ -\,2\,3\,4_5 \\ \hline 1\,2\,1_5 \end{array} \qquad \begin{array}{r} 4\,3\,6\,2_8 \\ -\,1\,7\,4\,5_8 \\ \hline 2\,4\,1\,5_8 \end{array} \qquad \begin{array}{r} E\,T\,5_{12} \\ -\,4\,7\,E_{12} \\ \hline 7\,2\,6_{12} \end{array}$$

Tables B.2b and B.3b provide the basis for multiplying numbers in bases five and two respectively. For example, the multiplication of 23 by 4 in base five proceeds as follows: From the entry in the "4" row and "3" column of Table B.2b we get $4\times3 = 22$. Writing down the rightmost 2 and "carrying" the other, we next obtain $4\times2 = 13$ from the "4" row and "2" column of the same table. Adding the carried 2 to this (in base five) gives 20. Hence, we have the following:

$$\begin{array}{r} 2\,3_5 \\ \times\ 4_5 \\ \hline 2\,0\,2_5 \end{array}$$

Similarly, the more difficult multiplication of $413_5$ by $23_5$ gives the following:

$$\begin{array}{r} 4\,1\,3_5 \\ \times\ 2\,3_5 \\ \hline 2\,2\,4\,4 \\ 1\,3\,3\,1 \\ \hline 2\,1\,1\,0\,4_5 \end{array}$$

Using Table B.3b, the product $1101_2 \times 1001_2$ is obtained as follows:

$$
\begin{array}{r}
1\ 1\ 0\ 1_2 \\
1\ 0\ 0\ 1_2 \\
\hline
1\ 1\ 0\ 1 \\
1\ 1\ 0\ 1 \\
\hline
1\ 1\ 1\ 0\ 1\ 0\ 1_2
\end{array}
$$

Some further examples of multiplication follow:

$$
\begin{array}{r}
1\ 2\ 0\ 1_3 \\
1\ 2\ 0_3 \\
\hline
1\ 0\ 1\ 0\ 2\ 0 \\
1\ 2\ 0\ 1 \\
\hline
2\ 2\ 1\ 1\ 2\ 0_3
\end{array}
\qquad
\begin{array}{r}
1\ 7\ 4\ 3_8 \\
3\ 5_8 \\
\hline
1\ 1\ 5\ 5\ 7 \\
5\ 6\ 5\ 1 \\
\hline
7\ 0\ 2\ 6\ 7_8
\end{array}
\qquad
\begin{array}{r}
1\ T\ 0\ E_{12} \\
4\ 0\ T_{12} \\
\hline
1\ 6\ 4\ 9\ 2 \\
7\ 4\ 3\ 8 \\
\hline
7\ 5\ T\ 0\ 9\ 2_{12}
\end{array}
$$

Although division is also performed in an analogous manner to that for the decimal system, it tends to be somewhat more confusing than the other operations. The following examples in binary and octal show how the operations proceed.

$$
\begin{array}{r}
\quad 1\ 0\ 1\ 1_2 \\
\hline
1\ 0\ 1_2 \ \big)\ 1\ 1\ 0\ 1\ 1\ 1_2 \\
\quad 1\ 0\ 1 \\
\quad\ \ 1\ 1\ 1 \\
\quad\ \ 1\ 0\ 1 \\
\quad\ \ \ \ 1\ 0\ 1 \\
\quad\ \ \ \ 1\ 0\ 1
\end{array}
\qquad
\begin{array}{r}
\quad 1\ 6_8 \\
\hline
7\ 2_8 \ \big)\ 1\ 4\ 6\ 5_8 \\
\quad\ \ 7\ 2 \\
\quad\ \ 5\ 4\ 5 \\
\quad\ \ 5\ 3\ 4 \\
\quad\ \ \ \ 1\ 1_8
\end{array}
\qquad \text{remainder } 11_8
$$

## Exercises

1. Construct addition and multiplication tables for arithmetic in the following bases: a) ternary b) septimal c) octal
   d) duodecimal e) hexadecimal

2. Perform the following additions:

   a) $11011_2 + 1101_2$ b) $1210_3 + 212_3$ c) $7156_8 + 4325_8$ d) $T1E4_{12} + 9ET8_{12}$

3. Perform the following subtractions:

   a) $10101_2 - 1011_2$ b) $4132_5 - 3413_5$ c) $6125_7 - 2416_7$ d) $1A4F_{16} - DB7_{16}$

4. Perform the following multiplications:

a) $1011_2 \times 101_2$ b) $4123_5 \times 341_5$ c) $4326_7 \times 314_7$ d) $1A43_{16} \times C5_{16}$

5. Perform the following divisions:

a) $121_3\overline{)2101_3}$ b) $413_5\overline{)2304_5}$

## B.3 Conversion of Integers

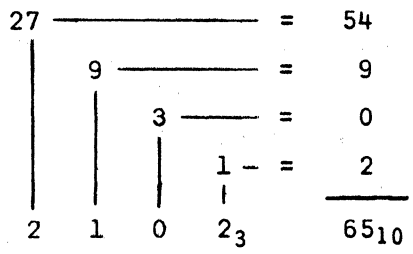
The techniques for conversion of an integer represented in one base to the representation in another base are based directly upon the principles of positional notation. To convert a number into decimal from some base other than ten, one needs only evaluate the decimal polynomial underlying the positional representation. This requires that the base be expressed in decimal and that the digits be converted to decimal, if necessary. For example, $431_5$ is converted to decimal as follows:

$$431_5 = 4\times5^2 + 3\times5^1 + 1\times5^0 = 4\times25 + 3\times5 + 1\times1$$
$$= 100 + 15 + 1 = 116_{10}$$

where all the arithmetic is performed in decimal. Similarly,

$$\begin{aligned} 101101_2 &= 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 \\ &= 1\times32 + 0\times16 + 1\times8 + 1\times4 + 0\times2 + 1\times1 \\ &= 32 + 0 + 8 + 4 + 0 + 1 \\ &= 45_{10} \end{aligned}$$

This conversion is sometimes more conveniently laid out as shown in the following example:




The evaluation of the decimal polynomial expansion requires fewer multiplications if the polynomial is first factored as follows:

$$( ( \ldots (d_n\cdot b+d_{n-1})\cdot b + \ldots + d_2)\cdot b + d_1)\cdot b + d_0$$

In this way only n multiplications and n additions are needed for an n+1 digit number. This method of evaluation may be laid out for simplicity as follows: The

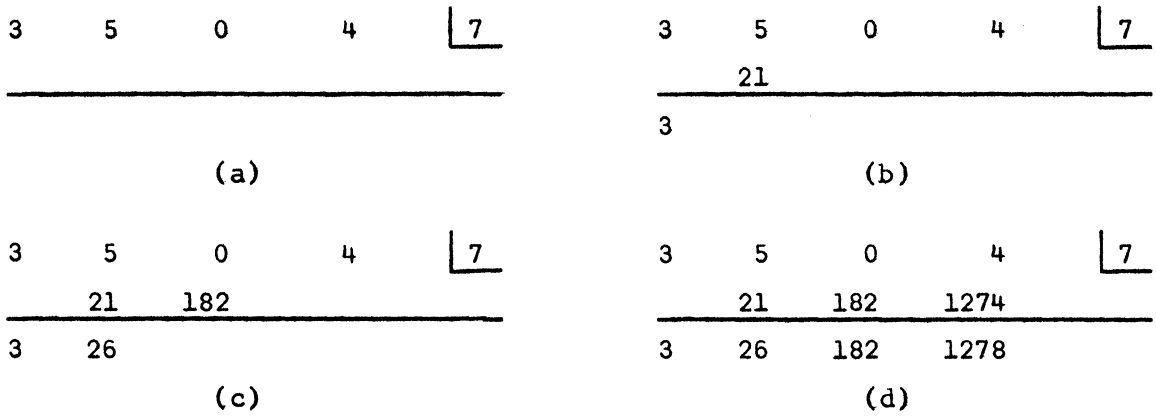



Figure B.1

digits are spread out horizontally with the radix set off to the right in an angle box, and a line is drawn under these leaving room for a row of numbers in between as shown for the number $3504_7$ in Figure B.1a. The first digit is copied below the line and this digit times the base is placed below the second digit as shown in Figure B.1b. The second digit is added to this product with the sum being placed below the line and this sum multiplied by the base is placed below the third digit as shown in Figure B.1c. This process is completed for all the digits as shown in Figure B.1d and the final number under the line is the decimal equivalent of the original number. Thus, the equivalence is as follows:

$$3504_7 = 1278_{10}$$

This method of evaluating a polynomial p(x) is called "synthetic division" in algebra since it produces the coefficients of the quotient polynomial q(x) which results when p(x) is divided by x-b as well as the remainder which is the value of p(b). This relationship is as follows:

$$p(x) = q(x)\cdot(x-b) + r$$

The first n numbers produced below the line are the coefficients of q(x) and the last number is the remainder. Remember that all of the arithmetic must be done in decimal for our particular application of this technique.

In the conversion of integers from decimal to other bases, the rationale is a bit more subtle. If an integer $d_n d_{n-1} \ldots d_1 d_0$ expressed in base b is divided by b, the remainder is $d_0$. This is made more clear by the following expansion:

$$\frac{d_n b^n + \ldots + d_1 b^1 + d_0 b^0}{b} = d_n b^{n-1} + \ldots + d_1 + \frac{d_0}{b}$$

One must get this same remainder (i.e. $d_0$) if the division is done in decimal. If one then divides the quotient by b, one gets $d_1$ as a remainder. Repeating this process one can find all the digits in the new representation. Suppose, for example, that we wish to convert $731_{10}$ to octal. We first divide 731 by 8 in decimal obtaining $731 = 91\times8 + 3$ so that $d_0 = 3$. Next we divide the quotient 91 from the first division by 8 getting $91 = 11\times8 + 3$ so that $d_1 = 3$. Similarly, we divide the second quotient 11 by 8 getting $11 = 1\times8 + 3$ so that $d_2 = 3$. Finally, we divide 1 by 8 getting $1 = 0\times8 + 1$ so that $d_3 = 1$. The quotient now being zero, all further digits in the octal representation will be zero. The base eight representation is therefore $1333_8$. The polynomial representation in base eight can also be produced by substitution in the above equations. First, one substitutes $11\times8 + 3$ for 91 in the equation resulting from the first division and obtains

$$731 = 91\times8 + 3 = (11\times8 + 3)\times8 + 3$$
$$= 11\times8^2 + 3\times8 + 3$$

Next one substitutes $1\times8 + 3$ for 11 in this expression giving:

$$731 = 11\times8^2 + 3\times8 + 3 = (1\times8 + 3)\times8^2 + 3\times8 + 3$$
$$= 1\times8^3 + 3\times8^2 + 3\times8^1 + 3\times8^0$$

Since this is the expansion of 731 as a polynomial in powers of eight, it is obvious that $731_{10}$ in octal is $1333_8$. In summary, converting $731_{10}$ to octal may be accomplished by repeated division by eight. The digits in the octal representation of $731_{10}$ are merely the remainders from the successive divisions, taken in reverse order. This division process can be arranged more concisely as shown in the following example. The representation of $731_{10}$ in base five is found by repeatedly dividing by five:

| | | |
|---|---|---|
| 5)731 | | |
| 5)146 | 1 | ↑ |
| 5)29 | 1 | \| |
| 5)5 | 4 | \| |
| 5)1 | 0 | \| |
| 0 | 1 | \| |

It is best to continue the division process until a quotient of zero is obtained so that the remainders all occur in a single column. Thus, the quinary representation is obtained by writing down these remainders in the order indicated by the arrow:

$$731_{10} = 10411_5$$

Conversion from decimal to binary requires only successive division by 2. For example, the conversion of $152_{10}$ to binary proceeds as follows:

$$
\begin{array}{rl}
2\overline{)152} & \\
2\overline{)76} & 0 \\
2\overline{)38} & 0 \\
2\overline{)19} & 0 \\
2\overline{)9} & 1 \\
2\overline{)4} & 1 \\
2\overline{)2} & 0 \\
2\overline{)1} & 0 \\
0 & 1
\end{array}
\qquad \uparrow
$$

Thus, we have the following result: $152_{10} = 10011000_2$

Actually all of the processes discussed above will work for conversion between any two bases. The polynomial evaluation techniques require that the multiplications and additions be done in the base <u>to which</u> conversion is being made, whereas, the repeated division techniques require that the division be done in the base <u>from which</u> conversion is being made. Since it is easier for most people to do the arithmetic in decimal, the techniques which allow this have been used so far in this discussion. To illustrate these other possibilities, $731_{10}$ is now converted to base five by converting its decimal polynomial representation to quinary and then evaluating it as follows:

$$
\begin{aligned}
731_{10} &= 7\times10^2 + 3\times10^1 + 1\times10^0 \quad \text{(all decimal)} \\
&= 12\times20^2 + 3\times20^1 + 1\times20^0 \quad \text{(all quinary)} \\
&= 12\times400 + 3\times20 + 1 \quad \text{(all quinary)} \\
&= 10300 + 110 + 1 \quad \text{(all quinary)} \\
&= 10411_5
\end{aligned}
$$

Similarly, $213_5$ may be converted to binary by successively dividing it by $2_5$ with the arithmetic being done in base five as follows:

$$
\begin{array}{rl}
2_5\overline{)213_5} & \\
2_5\overline{)104_5} & 0 \\
2_5\overline{)24_5} & 1 \\
2_5\overline{)12_5} & 0 \\
2_5\overline{)3_5} & 1 \\
2_5\overline{)1_5} & 1 \\
0_5 & 1
\end{array}
\qquad \uparrow
$$

Thus, we have the following result: $213_5 = 111010_5$

Because of the problem of doing arithmetic in bases other than ten, most people find it easier to convert between two bases neither of which are ten by converting from the given representation to decimal and then from decimal to the desired base. However, there are special cases, some of which arise frequently in computer work, in which a much more simple conversion is possible. The most common of these is binary-octal and octal-binary conversion. Since $8^n = 2^{3n}$, it is quite simple to replace the powers of eight in a polynomial such as

$$6\times8^3 + 3\times8^2 + 4\times8^1 + 5\times8^0$$

by powers of two:

$$6\times2^9 + 3\times2^6 + 4\times2^3 + 5\times2^0$$

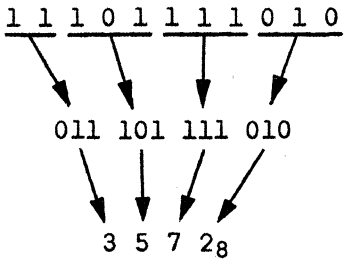
Now substituting the binary polynomial expansions of the digits 3, 4, 5 and 6 in this later polynomial gives:

$$\begin{aligned}(1\times2^2 + 1\times2^1 + 0\times2^0)\times2^9 + (0\times2^2 + 1\times2^1 + 1\times2^0)\times2^6 \\ + (1\times2^2 + 0\times2^1 + 0\times2^0)\times2^3 + (1\times2^2 + 0\times2^1 + 1\times2^0)\times2^0 \\ = \quad 1\times2^{11} + 1\times2^{10} + 0\times2^9 + 0\times2^8 + 1\times2^7 + 1\times2^6 \\ + 1\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0\end{aligned}$$

Comparing this with the original polynomial in powers of eight makes it obvious that

$$6345_8 = 110011100101_2$$

Hence, we see that conversion from octal to binary can be accomplished simply by replacing each octal digit by the three bit binary representation (leading zeros may be necessary to "fill in" all 3 bits) of the same number. Thus, if we memorize Table B.4 we find that we can convert from octal to binary by "inspection." Moreover, this procedure can be reversed: conversion from binary to octal can be accomplished merely by grouping the binary digits by threes (starting from the right and adding leading binary zeros if necessary) and then replacing each group by the corresponding octal digit. Thus, the conversion of $11101111010_2$ to octal is done as follows:




Thus, we have the following result: $11101111010_2 = 3572_8$

Similarly, conversion between binary and hexadecimal can be effected via the correspondence between 4 binary digits and 1 hexadecimal digit. This correspondence

Table B.4 Octal-Binary Correspondence

| Octal | Binary |
|---|---|
| 0 | 000 |
| 1 | 001 |
| 2 | 010 |
| 3 | 011 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

may be obtained from the hexadecimal and binary columns of Table B.1 by adding leading zeros to the binary representations so that they each have four bits.

For example, $110\ 0101\ 1100\ 1010_2 = 65CA_{16}$

and $DF13_{16} = 1101\ 1111\ 0001\ 0011_2$

Conversion between octal and hexadecimal may be easily accomplished by first converting to binary. The bits are then regrouped and the conversion completed.

As one might suspect, the principle underlying these conversions between binary and octal and between binary and hexadecimal can be extended to other cases where one base is an integral power of another, but they are not of interest in present computer applications. The only reason that the two given special cases are mentioned is that most people who work with binary computers find it easier and more concise to work with, say, 12 octal digits or 9 hexadecimal digits rather than 36 binary digits.

Exercises

1. Convert the following numbers to decimal:

   a) $1001_2$ b) $1001_3$ c) $1001_5$ d) $1001_8$ e) $423_5$

   f) $617_8$ g) $121_3$ h) $643_7$ i) $T2E_{12}$ j) $1A7_{16}$

2. Convert $797_{10}$ to:

   a) binary b) ternary c) quinary d) septimal

   e) octal f) duodecimal g) hexadecimal

3. Convert the following numbers to binary:

a) $ABCD_{16}$ b) $4321_{16}$ c) $4321_8$ d) $3027_8$

4. Convert the following binary numbers to octal and hexadecimal:

a) $1_2$ b) $1001_2$ c) $10001_2$ d) $110101101001_2$

## B.4 Fractions

So far our discussion of number systems has been confined to integers only. Fractions can also be handled in the various number systems in a manner analogous to decimal fractions where the digits to the right of the decimal point are the coefficients of negative powers of ten in a generalized polynomial. For example, just as the decimal fraction $.264301_{10}$ represents

$$2\times10^{-1} + 6\times10^{-2} + 4\times10^{-3} + 3\times10^{-4} + 0\times10^{-5} + 1\times10^{-6}$$

so also the octal fraction $.3701_8$ represents

$$3\times8^{-1} + 7\times8^{-2} + 0\times8^{-3} + 1\times8^{-4}$$

and .110101 represents

$$1\times2^{-1} + 1\times2^{-2} + 0\times2^{-3} + 1\times2^{-4} + 0\times2^{-5} + 1\times2^{-6}$$

or $$1/2 + 1/4 + 0/8 + 1/16 + 0/32 + 1/64 = 53/64$$

In "mixed" numbers, the radix point -- in decimal numbers, it's a decimal point; in octal numbers, it's an octal point; in binary numbers, it's a binary point, etc. -- serves simply to indicate which digit is the coefficient of the zeroeth power of the base in the polynomial

$$\ldots d_n b^n + d_{n-1} b^{n-1} + \ldots + d_1 b^1 + d_0 b^0 + d_1 b^{-1} + \ldots + d_m b^{-m} \ldots$$

The addition, subtraction, multiplication, and division of fractions in base eight, five, two, etc. is exactly analogous to that of decimal fractions. The following examples, showing a variety of such operations, should be carefully studied by the reader:

$$\begin{array}{r} {\scriptstyle 1\ 1\ 1\ 1\ \ \ \

$$
\begin{array}{r}
2 . 7\,5_8 \\
\times\ 3 . 4_8 \\
\hline
1\,3\,6\,4 \\
1\,0\,6\,7 \\
\hline
1\,2 . 2\,5\,4_8
\end{array}
\qquad\qquad
\begin{array}{r}
\phantom{1 . T_{12}\ )}\ 2\,3 . 9\,6_{12} \\
1 . T_{12}\ \overline{)\ 4\,2 . E\,5\,0_{12}} \\
\underline{3\,8}\phantom{\,.\,0_{12}} \\
6\,E\phantom{\,.\,0_{12}} \\
\underline{5\,6}\phantom{\,.\,0_{12}} \\
1\,5\,5\phantom{\,.\,0_{12}} \\
\underline{1\,4\,6}\phantom{\,.\,0_{12}} \\
E\,0\phantom{\,.\,0_{12}} \\
\underline{E\,0}\phantom{\,.\,0_{12}}
\end{array}
$$

Fractions which can be expressed exactly with a finite number of non-zero digits in one base may require a repeating pattern of digits in some other base. For example $1/2 = .5_{10}$, but to express 1/2 in base 5 one needs $.2222\ldots_5$, a situation similar to $1/3 = .3333\ldots_{10}$. Similarly, the fraction 1/7 is written $.\overline{142857}$ -- which means .142857142857142857... -- in decimal, but is simply .1 in septimal. Other examples are $1/3 = .\overline{3}_{10} = .4_{12}$ and $1/5 = .2_{10} = .\overline{0011}_2$.

Unlike the representation of integers, the representation of fractions may not be unique. Considering the decimal fraction represented by $.\overline{9} = .999\ldots$, it becomes quite clear that this fraction is "very close" to 1. Just how close it is can be shown as follows:

| | |
|---|---|
| Let | $x = 0.9999\ldots$ |
| Then | $10x = 9.9999\ldots$ |
| Subtracting, one gets | $9x = 9$ |
| And so | $x = 1$ |
| Thus | $1 = 0.\overline{9} = 0.9999\ldots$ |

A generalization of this shows that any decimal fraction terminating in an infinite sequence of 9's is equivalent to another decimal fraction with only zeros beyond the point where the 9's begin. For example, 0.20000... and 0.19999... represent the same decimal fraction. In a system having radix r, the digit representing r-1 plays a similar role to 9 in the decimal system in this regard. For example, .20000... and .17777... represent the same fraction (one-fourth) in the octal system.

The conversion of fractions from bases other than ten to decimal fractions may be performed by evaluating the polynomial, just as integers in bases other than ten are converted to decimal. For example,

$$
\begin{aligned}
.245_8 &= 2\times8^{-1} + 4\times8^{-2} + 5\times8^{-3} \\
&= 2/8 + 4/64 + 5/512 = 165/512 = .322265625_{10}
\end{aligned}
$$

The computations involved in evaluating fractional polynomials which terminate after a finite number of non-zero digits can be simplified if one observes that they may be factored. For example,

$$\begin{aligned} .245_8 &= 2\times8^{-1} + 4\times8^{-2} + 5\times8^{-3} \\ &= (2 + 4\times8^{-1} + 5\times8^{-2})\times8^{-1} \\ &= (2 + (4 + 5\times8^{-1})\times8^{-1})\times8^{-1} \end{aligned}$$

The arithmetic can now be organized as shown below on the left and the writing simplified as shown on the right.

```
8 ) 5 . 0 0 0
      . 6 2 5
  + 4 . 0 0 0
8 ) 4 . 6 2 5
      . 5 7 8 1 2 5
  + 2 . 0 0 0 0 0 0 0
8 ) 2 . 5 7 8 1 2 5
      . 3 2 2 2 6 5 6 2 5
```

```
8 ) 5 . 0 0 0
      . 6 2 5
8 ) 4 . 6 2 5
      . 5 7 8 1 2 5
8 ) 2 . 5 7 8 1 2 5
      . 3 2 2 2 6 5 6 2 5
```

Conversion of decimal fractions to other bases depends on multiplication in the same way that conversion of decimal integers to other bases depends on division. If one multiplies a fraction $.d_{-1}d_{-2}d_{-3}\ldots$ by the base b, the integer part of the product is the digit $d_{-1}$. Continuing this process $d_{-2}$ may be found by multiplying the fractional part of the above product by b. These same integer results must be produced if the multiplication is done in decimal. Thus, for example, $.149_{10}$ may be converted to octal as follows:

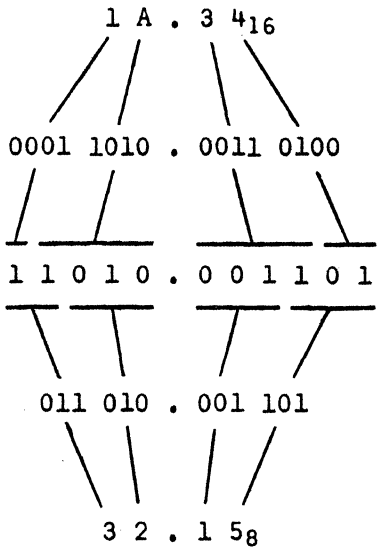
```
        . 1 4 9
          × 8
    ------------
1   . 1 9 2
          × 8
    ------------
1   . 5 3 6
          × 8
    ------------
4   . 2 8 8
          × 8
    ------------
2   . 3 0 4
```

Hence, $.149_{10} = .1142\ldots_8$. Note that only the fractional portion is multiplied by eight in each step and so it is safer to write the integer part of each product some distance to the left. These integer parts on the left are just the successive digits in the octal fraction, taken in the order indicated by the arrow. If the

fraction becomes zero, the conversion is complete. Otherwise, watch for a fractional portion which has occurred previously. The sequence of digits will repeat from this point on.

When "mixed" numbers containing both integer and fractional parts are to be converted from one base to another, the integer and fractional parts are converted separately using the techniques discussed above and the results are then combined.

Finally, since $2^{3n} = 8^n$ and $2^{4n} = 16^n$ even when n is negative, it follows that the octal-binary-hexadecimal conversions discussed above are also applicable to fractions. The important thing to remember is, in converting <u>from</u> binary to octal or hexadecimal, the "grouping" by threes or fours respectively <u>must</u> begin at the binary point and work both ways. In the following example a hexadecimal to binary and a binary to octal conversion are combined to achieve a hexadecimal to octal conversion:




<u>Exercises</u>

1. What fractions are represented by the following:

   a) $.01_{12}$ b) $.01_8$ c) $.01_{16}$ d) $.01_2$ e) $.01_5$

   f) $.101_2$ g) $.2_5$ h) $.8_{16}$ i) $.3_{12}$ j) $.75_8$

2. Add the following:

   a) $10110.0111_2 + 1011.1011_2$

   b) $25.64_8 + 6.15_8$

   c) $T.1E_{16} + 4.E5_{16}$

3. Subtract the following:

a) $101.1101_2 - 10.1010_2$

b) $7.6432_8 - .7516_8$

c) $A.BCD_{16} - .EF9_{16}$

4. Multiply the following:

a) $110.101_2 \times 1.01_2$

b) $2.43_5 \times 3.2_5$

c) $1.4E_{12} \times T2_{12}$

5. Divide the following:

a) $110.1_2\overline{)101.1101_2}$

b) $4.61_8\overline{)57.436_8}$

6. Express in the base indicated:

a) 1/3 in base 6

b) 1/8 in base 5

c) 2/3 in base 2

d) 1/4 in base 7

e) 1/2 in base 3

f) 1/6 in base 12

g) 1/2 in base 6

h) 1/8 in base 4

7. Convert to decimal:

a) $.25_8$ b) $.14_5$ c) $.4E_{12}$ d) $.A4_{16}$

8. Convert to base indicated:

a) $.2_{10}$ to base 5

b) $.25_{10}$ to base 8

c) .31 to base 3

d) .15 to base 8

e) .261 to base 12

f) .153 to base 16

9. Convert to decimal:

a) $110.1101_2$ b) $23.41_5$ c) $5.64_7$

10. Convert to the base indicated:

a) $10.375_{10}$ to base 2

b) 4.1875 to base 8

c) 7.415 to base 5

d) 14.629 to base 12

## Appendix C

## DEBUGGING AND TESTING TECHNIQUES

In this appendix, some techniques are discussed that may be helpful in verifying that a program is operating correctly or, if it is not, in finding the cause of the trouble. This discussion will concern only those errors occurring during execution, since the compiler diagnostics usually give sufficient information to pinpoint syntactical errors.

### C.1 Selecting Test Data

The selection of appropriate data sets for testing a routine can greatly ease the problem of detecting errors in the operation of the routine. Most important, the data sets must test all classes of data for which the routine was designed to work. This clearly does not mean that one must try all possible legal data sets, since this is usually an impossible task. On the other hand, one must supply sufficient sets of data so that all possible branches are tried. This requirement implies further that the data must be such that errors are not masked because they do not effect the results in that particular case. It is proper to assume that a routine is correct only when all different cases have been tried and the routine has produced the correct answers each time. Note that the requirement is that correct answers be produced, not just answers. This usually implies that you must check, by hand, that the answers produced for these test data sets are correct. In some cases, it is possible to write testing routines which cross check results on the machine and thus verify that a routine is producing correct results. For example, if one were writing a square root subroutine, it would be quite reasonable for a test routine to be written which called the subroutine with various values and then printed out the argument, the supposed square root, the square of this supposed root, and the difference between the argument and this square.

The test data sets should be as simple as possible. If, for example, a routine for adding n floating point mode numbers ($n \leq 100$) works properly for the sum of 1.6, 3.2, 4.7 and 5.0, one would expect that it would also work for the sum of 367.4129, 245.6031, 148.0001 and 106.3199. The use of the second set of numbers serves only to make the keypunching and the checking more difficult. Furthermore, if the routine works correctly for the sum of 4 and 5 numbers, there is no reason

to expect it not to work for the sum of 80 or 90 numbers, as long as sufficient space was reserved for their storage. Although that one set of data answers many questions about the operation of the routine, there are other checks that should be made. Data sets with negative numbers and n equal to one, zero and a negative number (if the routine is expected to do something in that case) should be tried. An error that also should be checked is the possibility that the routine is actually calculating the sum of n+1 numbers, rather than n. This will not be obvious on the first set of data since the extra number is the core constant, which is so small in magnitude that it would not normally effect the sum. If, however, the first data set had 5 numbers and a second data set had 4 numbers, the extra number will be the 5th number from the first data set and this error could be detected.

If the routine is to have many alternate paths, with several consecutive decisions (e.g. an income tax calculation where exemptions, marital status, withholding, deduction, and income level all affect the tax), there are two approaches that can be used. First, a small number of tests that checks all paths can be found. Typically, each data set would check paths in several sections of the program (e.g. one set may indicate single, 1 dependent, standard deduction, no withholding and lowest tax bracket). If all alternatives are checked and the routine is working correctly, this method will be quite efficient. If, however, there is a small error in the routine, it may be difficult to find this way, since the set of data for which the routine failed may be checking 5 or 6 different segments of the routine. If, on the other hand more data sets were used, each of which introduced only one new factor into the check, the error could be isolated more quickly.

## C.2 Interpreting Errors

When a routine is executing incorrectly, there are usually clues that indicate the cause of the trouble. In this section, we will discuss some of these symptoms and what they indicate.

Two common error messages are UNDERFLØW and ØVERFLØW, indicating the results of a floating point mode calculation were either too small or too large. Such an error may result from the use of data beyond the range of the machine, in which case the subroutines FTRAP. and FSPILL. can be used effectively. However, these errors are more commonly caused by the use of the core constant in a calculation. This may have resulted from a failure to initialize a variable or include it in the data set, using elements of a vector that were not defined (often because the terminating condition is too large, or from a spelling error which produces an undefined variable.

Another symptom that also may indicate the same problem is the printing of core constant or other very small or very large number.

The error messages TIME ESTIMATE EXCEEDED and PAGE CØUNT EXCEEDED usually occur as the result of improperly terminated loops. Most systems automatically insert a time limit (e.g. 30 seconds or 2 minutes) unless otherwise specified. Although some routines may require more than this time limit, one should remember that several million instructions can be executed in 30 seconds. If the page count has been exceeded, such a loop would contain print statements and the output should give a clue as to the location of the loop. The terminating condition of all loops and all transfers should be carefully checked. If the loop occurs in a subroutine, an error could be caused by the use of a floating point mode parameter on the calling list that corresponds to an integer mode formal parameter. Note that the floating point mode number 10. is, if interpreted as an integer, larger than $2^{34}$. No loop can be executed that number of times in less than several hours on existing machines. Another cause for failure to terminate properly is the use of .E. in the terminating test. Whenever possible, .GE. or .LE. should be used so as to avoid the possibility that equality is never exactly satisfied and the loop is not terminated.

Another symptom of improperly terminated loops is the destruction of low core (the system programs) and/or the destruction of the clock. This may be indicated on the printout by a message that the operator was forced to intervene or by excessive utility time. This type of error is usually caused by a loop that is not properly terminated in which values are calculated and stored in successive elements of a vector.

Exponentiation errors occur when the result of operations involving .P. would be too large or when a negative number is raised to a floating point mode power. This type of error can occur in the ARCCØS. and ARCSIN. subroutines if the parameter is greater than 1.0.

When routines in which normal mode is integer produce unreasonable (usually large) answers, one cause may be the failure to declare such subroutines as SQRT. and RANDØM. to be in floating point mode. The results of these routines are in floating point mode, and, if they are not so declared in the calling routine, the floating point mode number will be interpreted (incorrectly) as an integer.

The message WHAT WERE YØU DØING AT LØCATIØN XXXXX indicates an attempt to execute an illegal instruction or the core constant. This is often caused by transferring to an element of a label vector for which there is no correspondingly

labeled statement. This may also be caused by destroying parts of the program by using subscripts beyond the range for which space had been reserved.

The message ERRØR RETURN TØ SYSTEM is usually accompanied by a further message explaining the type of error. If this explanatory message is missing, the cause is often the omission of a statement label on the calling list of a subroutine for the error return. Another cause for this behavior is the execution of END ØF FUNCTIØN, which produces a call for ERRØR., not a return to the calling routine.

If a routine works correctly for the first set of data, but fails on subsequent data sets, the reason may be that certain variables are not properly initialized. This often does not affect the results on the first data set since core constant is so close to zero in floating point mode that the missing initialization is not detected. Also, if VECTØR VALUES is used for initialization of a variable, that declaration only initializes the variable when the routine is compiled, not for each data set and not each time a subroutine is called.

There is another common cause of premature termination of a program that can be very confusing. The message ALL INPUT DATA HAVE BEEN PRØCESSED may occur without the printing of any results, even though all paths through the program appear to have print statements. This behavior is usually the result of the omission of an asterisk at the end of a data set read using READ DATA or READ AND PRINT DATA.

## C.3 Some Tracing Techniques

Many of the errors discussed produce a printout of the location in memory of the instruction that caused the error. This information can help pinpoint the statement that is in error. The listing normally produced includes a map indicating the first location in memory reserved for each routine (system programs, main routine, user subroutines and subroutines automatically supplied). This enables one to find only the routine that was being executed when the error was made and very roughly where in the routine by interpolation. If more information is required, a listing of the object program can be obtained, which gives the machine language instructions produced by each MAD statement and their storage location relative to the beginning of that routine. This printout can be obtained by adding the phrase PRINT ØBJECT (after a comma) to the control card ($ card). Thus, for example, if the error occurred at location 10361 and the subroutine F. was stored in locations 10312 to 10426, the error occurred at location 10361 - 10312 = 47 in the listing of F..

Finally, let us consider the possibility that you have tried all the techniques in this appendix (as well as anything else you could think of) and still cannot find why the program isn't working correctly. (Furthermore, you cannot find your

instructor and the program is due in a few hours.) Also, you have tried simulating the computer, that is you have done the same calculations by hand that the computer should be doing. The next step is to get the computer to give you a detailed step by step printout of what it is doing. The system provides an option known as SUB TRACE to aid in debugging programs. If the phrase SUB TRACE is included (after a comma) on the control card ($ card), the system will provide a list of each entry to a subroutine (except input-output subroutines and those subroutines called by library subroutines) along with the normal printout. If this is not sufficient to enable you to locate the source of trouble, then print statements can be inserted at strategic spots in the program. Each such statement should be unique, so that you can tell which statement is being printed. This can be accomplished with a PRINT CØMMENT if you need only know whether a certain point was reached or by adding a constant to the output list in a PRINT RESULTS statement (e.g. PRINT RESULTS X, Y, 1.). Once it can be determined where the computer and hand calculations differ, finding the error is usually straightforward.

## C.4 Execution Time Error Messages

The following error messages which may occur during execution of a program are discussed in the text on the pages indicated.


| | |
|---|---|
| ALL INPUT DATA HAVE BEEN PROCESSED | 117 |
| BAD CARD | 117 |
| ERRØR RETURN TØ SYSTEM | 130 |
| EXP1 ANS TØØ LARGE | 131 |
| EXP2 NEGATIVE EXPØNENT | 131 |
| ØVERFLØW | 130 |
| PAGE CØUNT EXCEEDED | 150 |
| TIME ESTIMATE EXCEEDED | 150 |
| UNDERFLØW | 130 |
| WHAT WERE YØU DØING AT | 108 |

# Appendix D

## SELECTED MAD LIBRARY SUBROUTINES

This appendix lists a few of the most commonly used subroutines that are usually available in the MAD library. The contents of the library are not the same at all installations and thus some of the subroutines described herein may not be available at some places where MAD is used.

The identifier ERR is used in the typical calls to indicate a statement label used for error return statements. If such a label is omitted from the end of the calling list and the error condition is encountered, control is returned to the system and the program is terminated.

### D.1 Common Mathematical Functions

SQRT. ( <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | SQRT.(X), SQRT.(X,ERR), Y = SQRT.(X) |
| Error condition: | X .L. 0.0 |
| Value returned: | The square root of X ($\sqrt{X}$) |

EXP. ( <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | EXP.(X), EXP.(X,ERR), Y = EXP.(X) |
| Error condition: | X .G. 87.3 (EXP. would cause overflow) |
| Value returned: | 2.7182818 .P. X ($e^x$) |

ELØG. ( <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | ELØG.(X), ELØG.(X,ERR), Y = ELØG.(X) |
| Error condition: | X .LE. 0.0 |
| Values returned: | The natural logarithm of X (ln x) |

LØG10. ( <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Floating point mode function |
| Typical calls: | LØG10.(X), LØG10.(X,ERR), Y = LØG10.(X) |
| Error condition: | X .LE. 0.0 |
| Values returned: | The common logarithm of X ($\log_{10} x$) |

SIN. ( <f-p-exp> )

Subroutine class: Floating point mode function
Typical calls: SIN.(X), SIN.(X,ERR), Y = SIN.(X)
Error conditions: X .G. 6.8719477E10
Value returned: The sine of X where X is in radians (sin x)
Note: Although larger values are allowed, it does not make sense to use an X much larger than 1E6 since numbers that size have precision only to the nearest $1^{\circ}$. Values as large as 6E10 would involve rounding angles to the nearest 10000 degrees, giving useless answers.

CØS. ( <f-p-exp> )

Subroutine class: Floating point mode function
Typical calls: CØS.(X), CØS.(X,ERR), Y = CØS.(X)
Error conditions: X .G. 6.8719477E10
Value returned: The cosine of X where X is in radians (cos x)
Note: See Note above under sine function

ARCSIN. ( <f-p-exp> )

Subroutine class: Floating point mode function
Typical calls: ARCSIN.(X), ARCSIN.(X,ERR), Y = ARCSIN.(X)
Error condition: .ABS.X .G. 1.0
Value returned: The inverse sine Y (in radians) of X (arcsin x)
The value is in the range $-\frac{\pi}{2} \leq Y \leq \frac{\pi}{2}$

ARCCØS. ( <f-p-exp> )

Subroutine class: Floating point mode function
Typical calls: ARCCØS.(X), ARCCØS.(X,ERR), Y = ARCCØS.(X)
Error condition: .ABS.X .G. 1.0
Value returned: The inverse cosine Y (in radians) of X (arccos x)
The value is in the range $0 \leq Y \leq \pi$

ATAN. ( <f-p-exp> )

Subroutine class: Floating point mode function
Typical calls: W = ATAN.(Z)
Error condition: None
Value returned: The inverse tangent W (in radians) of Z
The resulting angle is in the range

$$-\frac{\pi}{2} \leq W \leq \frac{\pi}{2}$$

ATN1. ( <f-p-exp>, <f-p-exp> )

Subroutine class: Floating point mode function
Typical calls: W = ATN1.(Y,X)
Error condition: None
Value returned: The inverse tangent W (in radians) of $\frac{Y}{X}$

The resulting angle is in the range

$$0 \leq W \leq 2\pi$$

Note: ATN1. is an appropriate function for the calculation of the argument of a complex number.

TANH. ( <f-p-exp> )

Subroutine class: Floating point mode function
Typical call: W = TANH.(X)
Error condition: None
Value returned: The hyperbolic tangent of X (tanh x)

GAMMA. ( <f-p-exp> )

Subroutine class: Floating point mode function
Typical call: GAMMA.(X)
Error condition: None, but overflow or underflow will occur unless $-34.2 \leq X \leq 34.4$
Value returned: The gamma function $\Gamma(X)$, except that if X = 0. or a negative integer value, the value returned is .1E35

BSL1. ( <f-p-exp>, <int-exp> , <int-exp>, <f-p-var>, <int-exp> )

Subroutine class: Floating point mode function modifying other quantities (hybrid subroutine) Value returned used as flag only

Typical call: BSL1.(X,P,N,B,A)

Error condition: None

Calculations:

X is the argument of the Bessel function (x)

P is a code indicating which Bessel function is to be calculated as follows:

| | |
|---|---|
| 1 | $J_n(x)$ |
| 2 | $I_n(x)$ |
| 3 | $K_n(x)$ |
| 4 | $Y_n(x)$ |
| 5 | $e^{-x}Y_n(x)$ |
| 6 | $e^{-x}K_n(x)$ |

N is the order of the Bessel function (n)

B is the value of the Bessel function when a successful calculation has taken place

A is the accuracy of the result produced where the result contains 27-A bits of accuracy and $0 \leq A < 27$

1 is the functional value returned if the calculations are successful

2 is the functional value returned if the input parameters would produce a result too large for the computer

## D.2 Subroutines for the Control of Overflow and Underflow

FTRAP.
NTRAP.

Subroutine class: Procedures
Error condition: None
Action: These subroutines have no parameters and cause no immediate action. They set a location in the system portion of memory that handles floating point mode underflow (i.e. non-zero results of calculations less, in magnitude, than approximately .1E-37). Ordinarily, underflow produces an error return to the system. If FTRAP. is called before an arithmetic operation that produces underflow is executed, the result will be set to zero and execution will continue. A call for NTRAP. nullifies the effect of FTRAP., i.e. restores underflow as an error. Since these subroutines set a location in the system portion of memory, the effect of underflow is determined by the last call for these routines (see also FSPILL. and RSPILL. below), even if that call is within a different routine in that program.

FSPILL. (<label>)
RSPILL.

Subroutine class: Procedures
Typical call: FSPILL.(S), RSPILL
Error condition: None
Action: These routines handle floating point overflow, (i.e. magnitude of the result greater than approximately .17E39) as well as underflow. FSPILL. has the same effect on underflow as FTRAP. and RSPILL. has the same effect as NTRAP. In addition, FSPILL. sets the location in the system portion of memory that is checked when overflow occurs. Normally that location indicates an error return to system. After FSPILL.(S) is called, that location indicates a transfer to the statement labeled S. RSPILL. nullifies the effect of FSPILL., i.e. restores both overflow and underflow as errors. Note that the location returned to is determined by the last call for FSPILL. in that program set. If that call is in a different routine from the one in which overflow occurred, control will be transferred to the routine containing the call for FSPILL. which was executed last.

## D.3 Random Number Generation Subroutines

### RANDØM. ( <f-p-var> )

Subroutine class: Floating point mode function
Typical call: Y = RANDØM.(R)
Error condition: None
Value returned: A pseudo random floating point number, Y, chosen from a uniform distribution in the range

$$0.0 \leq Y \leq 1.0$$

The parameter R is used to propagate a pseudo-random number sequence. A non-zero positive initial value may be assigned to R by the user before RANDØM. is called. Any such non-zero positive value will cause the routine to generate a pseudo-random sequence which is determined by that value. If the initial value of R is zero, the routine logically combines certain locations in low core, including the clock and various buffers, to obtain a starting point so that the probability of repeating the same sequence of random numbers on successive uses of this initialization is very small (less than .0000002). The argument R must not be a constant. For example, the call Y = RANDØM.(0.0) would cause the constant 0.0 to be modified each time RANDØM. is called and thus 0.0 would no longer be zero elsewhere in the routine.

Note: RANDØM. employs the thoroughly tested power residue method of random number generation. The periodicity of this method is $2^{35} - 1$.

### RANDND. ( <f-p-exp>, <f-p-exp>, <f-p-var> )

Subroutine class: Floating point mode function
Typical call: RANDND.(M,S,X), Y = RANDND.(M,S,X)
Error condition: None
Value returned: A pseudo-random floating point number chosen from a normal distribution with mean M and standard deviation S. The parameter X is used to propagate the pseudo-random sequence. This function calls RANDØM. using this X as the argument and then transforms the result to obtain a normal distribution. As long as different arguments (e.g. X) are used, the calling of RANDND. will not affect the sequences produced by other uses of RANDØM.

## D.4 Complex Arithmetic Subroutines

CMADD. ( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

Subroutine class: Procedure
Typical calls: CMADD.(A,B,C,D,E,F), EXECUTE CMADD.(A,B,C,D,E,F)
Error condition: None
Calculations: E + iF = (A + iB) + (C + iD)

CMSUB. ( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

Subroutine class: Procedure
Typical calls: CMSUB.(A,B,C,D,E,F), EXECUTE CMSUB.(A,B,C,D,E,F)
Error condition: None
Calculations: E + iF = (A + iB) - (C + iD)

CMMUL. ( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

Subroutine class: Procedure
Typical calls: CMMUL.(A,B,C,D,E,F), EXECUTE CMMUL.(A,B,C,D,E,F)
Error condition: None
Calculations: E + iF = (A + iB) · (C + iD)

CMDIV. ( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

Subroutine class: Procedure
Typical calls: CMDIV.(A,B,C,D,E,F), EXECUTE CMDIV.(A,B,C,D,E,F)
CMDIV.(A,B,C,D,E,F,ERR), EXECUTE CMDIV.(A,B,C,D,E,F,ERR)
Error condition: C = D = 0.0
Calculations: E + iF = (A + iB) / (C + iD)

CMSQRT. ( <f-p-exp>, <f-p-exp>, <f-p-var>, <f-p-var> )

Subroutine class: Procedure
Typical call: CMSQRT.(A,B,C,D), EXECUTE CMSQRT.(A,B,C,D)
Error condition: None
Calculations: The result C + iD is the complex square root of A + iB

CFAD. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )
CFSB. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )
CFMP. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )
CFDP. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )

These subroutines are available in some systems in place of the corresponding CMxxx. routines. The typical call of the routine

CFyy.(A(1),B(1),C(1))

would correspond to

CMxxx.(A(1),A,B(1),B,C(1),C)

## D.5 Double-Precision Arithmetic Subroutines

DFAD. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )
DFSB. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )
DFMP. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )
DFDP. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )
SFDP. ( <f-p-vector-ref>, <f-p-vector-ref>, <f-p-vector-ref> )

| | |
|---|---|
| Subroutine class: | Procedures |
| Typical calls: | DFAD.(A,B,C), EXECUTE DFAD.(A,B,C)<br>DFSB.(A,B,C), EXECUTE DFSB.(A,B,C)<br>DFMP.(A,B,C), EXECUTE DFMP.(A,B,C)<br>DFDP.(A,B,C), EXECUTE DFDP.(A,B,C)<br>SFDP.(A,B,C), EXECUTE SFDP.(A,B,C) |
| Error condition: | In DFDP. and SFDP. only, if the high order word of the divisor is zero an error return to system is taken unless DCEXIT. has been called. (See below.) |
| Calculations: | These subroutines perform the double-precision floating point operations of addition, subtraction, multiplication and division with A and B as the operands and C as the result (see note below). SFDP. ignores the low-order word of the divisor by assuming it to be zero. |

DCEXIT. ( <label-var> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | DCEXIT.(L), EXECUTE DCEXIT.(L) |
| Error condition: | None |
| Action: | After execution of this procedure, any attempt to use a divisor whose high order word is zero in DFDP. or SFDP. will cause a transfer to the statement specified by L. |

DSQRT. ( <f-p-vector-ref>, <f-p-vector-ref> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | DSQRT.(A,B), EXECUTE DSQRT.(A,B) |
| Error condition: | None |
| Calculations: | The result B is the double-precision floating point mode square root of the argument A with the sign of the argument, i.e., the square root of a positive number is positive and of a negative number is negative. (See note below.) |

Note: In the above routines, the double-precision floating point numbers must be stored in adjacent locations of a floating point mode vector, say V for example. If the high-order (i.e. most significant) word is stored in V(I), then the low-order word must be in V(I-1). The low-order word has the same sign and a characteristic which is 27 (base 10) less than the high-order word. Reference is made in the calling list only to the high-order words of the parameters. The high-order word of a result is always normalized.

DPFA. ( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp> )
DPFM. ( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp> )
DPFDV.( <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp>, <f-p-exp> )

| | |
|---|---|
| Subroutine class: | Procedures |
| Typical calls: | DPFA.(A,B,C,D,E,F), EXECUTE DPFA.(A,B,C,D,E,F)<br>DPFM.(A,B,C,D,E,F), EXECUTE DPFM.(A,B,C,D,E,F)<br>DPFDV.(A,B,C,D,E,F), EXECUTE DPFDV.(A,B,C,D,E,F)<br>DPFDV.(A,B,C,D,E,F,ERR),<br>EXECUTE DPFDV.(A,B,C,D,E,F,ERR) |
| Error condition: | In DPFDV. only, C = 0.0 and D = 0.0 |
| Calculations: | These subroutines perform double-precision floating point addition, multiplication, and division, respectively. A is the high-order word of the first operand and B is the low-order word of the first operand. C and D are the high and low-order words of the second operand and E and F are the high and low-order words of the result. |

Note: A simple equivalence exists between certain calls for the two groups of double-precision floating-point subroutines given in this section. For example, the call

DPFA.(A(1),A,B(1),B,C(1),C)

is equivalent to

DFAD.(A(1),B(1),C(1))

## D.6 Subroutines to Control Subroutine Tracing Feature

ONTRC.
OFFTRC.

| | |
|---|---|
| Subroutine class: | Procedures |
| Typical calls: | ONTRC., EXECUTE ONTRC.<br>OFFTRC., EXECUTE OFFTRC. |
| Error condition: | None |
| Action: | These subroutines make it possible to control the system subroutine tracing feature during execution. They are not effective unless SUB TRACE appeared on a system control card. Execution of ØFFTRC. suppresses the printing of the subroutine trace, and execution of ØNTRC. reactivates the subroutine trace. Unless ØFFTRC. has been called the subroutine tracing feature is in operation. Note: See Appendix C for a discussion of the SUB TRACE feature. |

## D.7 Initialization Subroutines

### ZERØ. ( L{ <var> or <var>...<var> } )

Subroutine class: Procedure
Typical call: ZERØ.(A,B(1)...B(193))
Error condition: None
Action: The value zero is stored in the elements specified in the list (e.g. A, B(1), B(2), B(3), ..., B(193) in the typical call). Since the Boolean, floating point and integer zeros are identical, parameters of different modes may be included in a single call.

### SPRAY. ( <exp>, L{ <var> or <var>...<var> } )

Subroutine class: Procedure
Typical calls: SPRAY.(6.3,X...X(6))
SPRAY.(X+3.0,A,B,C,D)
Error condition: None
Action: The value of the expression appearing as the first parameter is stored in the elements specified in the list. Conversion of mode does not take place, and so the elements of the list should normally be of the same mode as the expression.

## D.8 Subroutines for Conversion between Binary Coded Decimal and Binary

### BCDBN. ( <int-exp> )

Subroutine class: Integer mode function
Typical calls: BCDBN.(N), I = BCDBN.(N)
Error condition: None
Value returned; The integer equivalent to the BCD argument, N. Blanks in N are ignored.
Example: N = 0 0 1 8 4 7 (BCD)
= 000001100407 (octal)
I = 000000003467 (octal)

### BNBCD. ( <int-exp> )

Subroutine class: Integer mode function
Typical calls: BNBCD.(I), N = BNBCD.(I)
Error condition: None
Value returned: The BCD equivalent to the integer argument, N, with leading zeroes added if necessary. I must be no greater than 999999.
Example: I = 000000000136 (octal
N = 000000001104 (octal)
= 0 0 0 0 9 4 (BCD)

# Appendix E

## FORMATTING AND INPUT-OUTPUT

Some input and output statements which are more versatile than those given in Chapter Three are presented in this Appendix. Most of these statements allow more flexibility in the listing of the items to be input or output and operate under the control of format specifications with which one can exactly prescribe the use of each character position on input or output. Other statements provide for the reading and writing of binary tapes, and the control of tape positioning and recording densities.

### E.1 Formatted Input and Output Statements

Each of the input and output statements described in this section includes a format expression, the value of which specifies the exact way in which the input appears on cards or the form that the output is to take on cards or on paper. In input statements, the format expression is followed by a list of the input elements whose values are to be read. In output statements, the format expression is followed by a list of output elements the values which are to be printed or punched. Although these lists may be similar to the lists used in the PRINT RESULTS statement (except that constants and compound expressions may not be used on input lists), they may also involve iterations similar to those used in the increment form of the THRØUGH statement. The discussion of these iteration elements will be given in Section E.4 where it will be more meaningful.

We will first describe the behavior of the five input-output statements and then go into the details of format specifications. The two input statements are:

READ FØRMAT <format-exp>#, <format-input-list>
LØØK AT FØRMAT <format-exp>#, <format-input-list>

where we use the

Definition of format input list, format input element and input variable:

<format-input-list> is L{ <format-input-element> } or <null>
<format-input-element> is <input-var> or <input-var> ... <input-var> or
<input-var> ,..., <input-var> or
( <v> = <i> , <s> , <terminate> , <format-input-list> )
<input-var> is <a-var> or <bool-var>

The metavariables <v>, <i>, <s> and <terminate> are defined in Section 6.3 as part of the discussion of the increment form of the THRØUGH statement.

Each of these input statements reads a card (possibly more than one on the READ FØRMAT statement), using the information given by the format expression, and stores the data on these cards in the locations reserved for the variables indicated on the list. The difference between the two statements (in addition to the limitation of one card on LØØK AT FØRMAT) is in what happens on the next input statement. The LØØK AT FØRMAT statement leaves the input tape so that the next input statement reads the same card again. The READ FØRMAT statement (like READ DATA and READ AND PRINT DATA) advances the input tape so that the next data set will be read by the next input statement.

The three output statements are:

PRINT FØRMAT <format-exp>#, <format-output-list>
PRINT ØN LINE FØRMAT <format-exp>#, <format-output-list>
PUNCH FØRMAT <format-exp>#, <format-output-list>

where we have the

Definition of format output list, format output element, output variable, output expression and output block:

<format-output-list> is L{ <format-output-element> } or <null>
<format-output-element> is <output-var> or <output-exp>
or <output-block> or
( <v> = <i>, <s>, <terminate>, <format-output-list> )
<output-var> is <a-var> or <bool-var>
<output-exp> is <a-exp> or <bool-exp>
<output-block> is <output-var> ... <output-var> or
<output-var> ,..., <output-var>

The PRINT FØRMAT statement results in the values of the expressions indicated on the output list being printed according to the specified format as part of the normal output listing for the program. Variable names are not printed (as they are with PRINT RESULTS) unless the format specification includes the variable name (see below). The PRINT ØN LINE FØRMAT behaves in the same way as PRINT FØRMAT except that the printing is done by the on-line printer and is not included with the normal output listing. This statement is normally used for messages to the operator. The PUNCH FØRMAT statement is used to punch results on cards, rather than to print them.

## E.2 The Format Specification

A format expression may be the actual format specification followed by an asterisk (*) and enclosed in dollar signs ($), or a reference to a vector that contains the format specification and the asterisk. A format specification describes a set of consecutive lines (i.e. lines of print or cards) which are to be read or written. Although more complex structures are sometimes used, in most cases the format specification consists of a list of line specifications separated by slashes (/). The asterisk (*) is used to indicate the end of the format specification. A line specification describes how a single line is to be read, printed or punched in terms of field specifications which may, for example, prescribe how a particular number is to be read or written. Each field specification describes a group of consecutive columns in a line and the successive field specifications describe the line from left to right (except for the use of the tabulation (T) specification which will be discussed later).

Although the general definitions and discussion of formatted input and output are given here, formal definitions of the fields which may be used and detailed description of the action they cause will be left until the next two sections. Definition of format expression, format specification, line specification, and multiplicity:

```
<format-exp>    is  <int-vector-ref>  or  $<format-spec>*$
<format-spec>   is  <line-spec>  or  <mult> ( <format-spec> )
                    or  <format-spec> / <format-spec>
                    or  <format-spec> , <format-spec>
<line-spec>     is  L{ <field-spec>#  or  <T-spec># }  or  <null>
<mult>          is  <deci-con>  or  <format-var-use>#  or  <null>
```

A slash (/) in a format specification (except within the H fields discussed below) indicates that the specification of a new line is to follow. The use of null line specifications causes lines to be ignored on input and left blank on output. Null lines are indicated by consecutive slashes (//) or slashes at the beginning or end of the specification.

There are seven basic types of fields that can appear on a card or paper. Each of these fields is specified by a single letter (indicating the type of field) and, except for H fields, is followed by an integer indicating the width of that field (i.e. the number of columns specified). The characters for the basic field

specifications are as follows:

S Skip (or blank)

I Integer (used for both integer and Boolean data)

E Floating point mode number, with exponent or characteristic (e.g. 1.2E3)

F Floating point mode number, without exponent (e.g. 7.63)

K Octal number (can be any mode internally)

C Character constant or string

H Hollerith character string included in the format specification

Boolean values are treated as integers in a format specification, and are printed and punched as 0 (not 0B for false) and 1 (not 1B for true). Formally we have the

Definition of basic field specification:

<basic-field-spec> is <S-field-spec> or <I-field-spec>
or <E-field-spec> or <F-field-spec>
or <K-field-spec> or <C-field-spec>
or <H-field-spec>

If the format specification is used for printing (not for punching), the first character of each line is used to control the printer carriage (just as in PRINT CØMMENT) and is thus not printed. The actions caused by the various carriage control characters are listed in Table 3.4 on page 122. The carriage control character is usually supplied with a skip (S) field (which gives a blank) or the first character of a Hollerith (H) field. The use of every column of a card or a line of print must be described in a line specification, except that if less than a full card or a full print line is described, the remainder of the line is ignored on input or filled with blanks on output. The print line, including the carriage control character, normally has 132 characters for output listing, but only 120 characters for on-line output.

As an example, consider the simple input format specification

```
S10, I6, C5
```

which might be used in statements such as

```
READ FØRMAT  $S10,I6,C5*$, A, B
```

or

```
LØØK AT FØRMAT  F, A, B
```

where the integer mode vector F contains this format specification stored as a character string. This format specifies how one card is to be read. The first 10 columns of the card are to be ignored, the next 6 columns (11-16) contain an integer,

and the next 5 columns (17-21) are to be interpreted as a character constant. The input list in each of the statements specifies that the first constant (the integer) is to be stored in A and the second constant in B. Since the format only specifies 21 columns, the remainder of the card is to be ignored and so this specification is equivalent to S10, I6, C5, S59. Note that each printed output line must contain its own carriage control character and thus, if the slash is used on printing, the first character specified following the slash is used for carriage control. If the line specification is <null> a line is simply skipped. Thus, the specification

S6, I3 / S6, C4, S6, I9 // S12, I7

refers to 4 lines or cards, the first of which is specified by S6, I3; the second is specified by S6, C4, S6, I9; the third, being null, is skipped; and the fourth is specified by S12, I7.

The lists following the format reference in the input and output statements consist of a set of items to or from which data is to flow. For input, these items must be variables, blocks of variables, or iteration elements, whereas for output, they may also be constants or expressions. Except for H and S fields, each basic field specification corresponds to one item on the list, where each variable indicated in a block or in an iteration element, is considered an item. During the transmission of data, the input and output subroutines scan both the list and the specification simultaneously and corresponding items on the list and the specifications are associated. Since no conversion of mode takes place with formatted input and output, errors may result in storage on input or in printing on output if the mode of the input or output item is not compatible with the field specification. The input or output statement is terminated when the list is exhausted. What happens when there are items remaining on the list when the specification is exhausted (i.e. the list contains more items than the format specification), will be discussed in Section E.4.

## E.3 Field Specifications

The basic field specifications for skip (S), integer (I), character (C), and octal (K) fields are the most simple, consisting of the letter specifying the type of the field followed by a specification of the width of the field. They will be discussed first, with discussion of the floating point (E and F) and Hollerith (H) field specifications following.

Definitions of S, I, C, and K field specifications and field width:

<S-field-spec> is S <width>
<I-field-spec> is I <width>
<C-field-spec> is C <width>
<K-field-spec> is K <width>
<width> is <deci-con> or <format-var-use>#

In the discussion which follows, we will use w to designate the numeric value of the width, as opposed to the metalinguistic variable <width> which is used in defining the language.

The action caused by a skip (S) field specification is very simple. On input, the next w characters in the line being specified are ignored, regardless of what characters are in these positions. On output, the next w positions in the line have blanks inserted in them.

The integer (I) field specifications are used to read or write integers. Since no conversion of mode takes place on input and output, items on the input and output lists which correspond to I fields should usually be in integer mode although Boolean input and output is also handled with I fields. On input, an integer is read from the field and stored in the corresponding input item. Blanks are ignored, and the field must contain no non-blank characters other than plus (+), minus (-), and the digits. The plus (+) sign may be omitted, but the minus (-) sign must be included to designate a negative number. The sign must be punched to the left of the first digit. An all blank field will be read as a negative zero (-0). On output, the value of the next output item, interpreted as an integer, is printed in the field, right justified (with blanks inserted on the left to fill the field). A minus (-) sign is printed if the output item is negative, but no plus (+) sign and no decimal point is printed. If the number of columns specified is insufficient to hold the number (e.g. attempting to print -73 or 6412 with an I2 format), an error occurs and the program is normally terminated (see Section E.8). Some examples of

input and output under an I6 field specification follow:

| Card Field | Internal | Print Field |
|---|---|---|
| Δ2Δ1Δ3 | 213 | ΔΔΔ213 |
| -ΔΔΔΔ1 | -1 | ΔΔΔΔ-1 |
| ΔΔΔΔΔΔ | -0 | ΔΔΔΔ-0 |

With a basic character (C) field specification, a maximum of six (6) BCD characters may be read or written. Blanks are significant in C fields and it is assumed that input and output items corresponding to C fields are in integer mode. The data is stored internally in BCD form. On input, if less than 6 characters are specified, these characters are stored in the word left justified (with blanks added at the end). If more than 6 characters are specified, those characters after the first 6 are ignored. On output, character constants are left justified, and thus blanks are added on the right when the field is longer than 6 columns. For example, if the first ten columns of a card contained

ABCDEFGHIJ

the input statement

READ FØRMAT $C2,C7,C1*$, A, B, C

would cause the following storage:

A = ABΔΔΔΔ B = CDEFGH C = JΔΔΔΔΔ

where the I is lost because B can only hold 6 characters. Using a different format specification, the output statement

PRINT FØRMAT $S2,C3,C4,C2,C6*$, A, B, C, B

would then cause the following line to be printed

ΔABΔCDEFJΔCDEFGH

Octal (K) field specifications are used to obtain conversion between binary representation internally and octal representation externally. The items on the input and output lists may be of any mode, conversion being independent of this mode. On input, blanks are ignored and the field must contain no non-blank characters other than plus (+), minus (-), and the octal digits. The plus (+) sign may be omitted, but the minus (-) may be included to designate a negative number. Octal points are not allowed, the octal integer being pushed to the right in the computer word with zeros inserted on the left. On output, the last w octal digits of the computer word are printed. Any digits preceding these w are lost, zeros on the left are not suppressed and signs (+ or -) are never printed. Thus, if a negative number is being output with w less than twelve, the sign information will be lost. Some examples of octal input under a K10 field and output of the same data under both a

K12 and a K10 field specification follow:

| Card Field | K12 Print Field | K10 Print Field |
|---|---|---|
| Δ6Δ4Δ2Δ3ΔΔ | 000000006423 | 0000006423 |
| -157ΔΔΔΔΔΔ | 400000000157 | 0000000157 |
| ΔΔΔΔΔΔΔΔΔΔ | 000000000000 | 0000000000 |

The floating point (E and F) field specifications must specify the number of decimal places that follow the decimal point in addition to the field width. If this number is zero the specification may be omitted. Thus, the Definition of E and F field specification and decimal places:

<E-field-spec> is E <width> or E <width> • <places>
<F-field-spec> is F <width> or F <width> • <places>
<places> is <deci-con> or <format-var-use>#

We will use p to designate the numeric value of <places>. On output, p places will be printed after the decimal point, but on input this value is used only if an explicit decimal point does not occur in the field. The number of places p must be less than the field width and additional positions must be allowed for decimal points, signs and exponents. Data specified in an E or F field is assumed to be stored in floating point mode internally. As is the case for integers, plus signs may be omitted on input and are always omitted on output. Minus signs, however, must be included on input and are always printed or punched. On output, the numbers are right justified in the field--with blanks inserted on the left.

For input with either an E or an F field specification, a number may be punched with or without an exponent. This exponent may be of the usual form

$$E\ \{\ +\ \underline{or}\ \langle null\rangle\ \underline{or}\ -\ \}\ S_1^2\{\ \langle digit\rangle\ \}$$

where the plus sign may be omitted and one or two digits may be used. However, if the sign appears explicitly (i.e. the sign is punched) then the "E" may be (but need not be) omitted, so that

$$\{\ +\ \underline{or}\ -\ \}\ S_1^2\{\ \langle digit\rangle\ \}$$

is also acceptable. This exponent must, of course, follow the mantissa in the field and room for an exponent must be allowed for in the field specification if any are to occur in the input data. If the mantissa of the number has an explicit decimal point (i.e. if the decimal point is punched), then the "places" specification is ignored. If a decimal point does not appear in the mantissa, then one is assumed to precede the last p digits of the mantissa. Characters other than digits, signs, decimal points and "E" in these fields will cause an error return to system and even

these characters must be used properly (e.g. a field with two decimal points is in error). Some examples of input card fields and the floating point number which would be read with either an E10.3 or an F10.3 field specification are as follows:

| Card Field | Number Read |
|---|---|
| ΔΔ-13.67ΔΔ | -13.67 |
| .136Δ7ΔΔΔΔ | 0.1367 |
| 13ΔΔΔΔΔΔ67 | 1.367 |
| 1367.E+3ΔΔ | 1367000.0 |
| Δ-1367E+03 | -1367.0 |
| .13Δ67+3ΔΔ | 136.7 |
| Δ-1367-4ΔΔ | -0.0001367 |

In output under an F field specification with p greater than zero, a decimal point is printed or punched followed by the fractional part of the number rounded to p places. When p is zero, the decimal point is not printed or punched and the number is rounded to the nearest integer. The integer part of the number, with leading zeros suppressed (replaced by blanks) and a minus (-) sign preceding it in the case of negative numbers, is printed or punched immediately ahead of the decimal point, if any. The printing is right justified in the field (with blanks to the left). If the resulting number requires more than w characters, including sign and decimal point, an error return to system normally results. Note that F6.3 allows only two places to the left of the decimal point including a possible minus (-) sign. Some examples of output using a field specification of F6.3 are as follows:

| Number | Output |
|---|---|
| 3.6148 | 3.615 |
| 67.0 | 67.000 |
| -.03 | -.030 |
| .00012 | .000 |
| 428.1 | Error |

In output under an E field specification, the characteristic always occupies the last four positions of the field in the following form

$$E\ \{\ \langle\text{blank}\rangle\ \underline{\text{or}}\ -\ \}\ \underline{S}_2^2\{\ \langle\text{digit}\rangle\ \}$$

The mantissa, which is always between 0.1 and 1.0 unless the number is zero or the field specification is modified, is rounded to p places and printed or punched immediately ahead of the characteristic. A decimal point and in the case of negative numbers, a minus (-) sign precedes the mantissa. Since six spaces must be allowed in the field for characteristic, decimal point and possible sign, one must add this six to the number of significant places, p, which are desired to obtain the minimum

field width. Some examples of E field outputs under the specification E10.3 are:

| Number | Output |
|---|---|
| 6.3 | 0.630E 01 |
| 6.341 | 0.634E 01 |
| -.007 | -0.700E-02 |

The Hollerith (H) field specification is used primarily for the output of character strings for headings and other situations where the string does not change. They can, however, be used for input also, as will be explained later. Although the C field specification may be used to print (or punch) characters, these characters must have been input as data, preset with vector initialization or obtained in some manner. The Hollerith field, which allows the characters string itself to be included in the format specification, may be of two forms. First, the letter H may be preceded by an integer specifying the length of the field. The characters to be printed then follow the H. Thus, a field 7HØNE TWØ would request the printing of the words ØNE TWØ. Instead of preceding the H by the length of the field, the field may be delimited by any character. Thus, if H is not preceded by an integer, the first character following the H is treated as a delimiter. The field to be printed then consists of all characters following the delimiter up to (but not including) the next appearance of that delimiter. Since all format specifications are stored as vectors of character constants, dollar signs must be denoted by $$ in all Hollerith fields. Thus, four equivalent Hollerith fields are:

```
7HØNE TWØ
H'ØNE TWØ'
HHØNE TWØH
H$$ØNE TWØ$$
```

Definition of H character, delimiter and H field specification:

<H-char> is { <character> not $ } or $$

<d> is <H-char>$_1$

<H-field-spec> is <width> H $\underline{S}_w^w$ { <H-char> } or

H <d> $\underline{S}_0^*$ { <H-char> not <d> } <d>

Note that each reference to <d> in the definition of the H field specification denotes the same H character. Blanks are significant in Hollerith fields and must be included in the count of the field length. The comma following a Hollerith field in a format specification may, but need not, be omitted.

In output, the characters of the Hollerith field specification are simply copied onto the output line (or card) in the next w positions (where w is obtained implicitly in the delimiter form of the Hollerith field).

A Hollerith field may also be used in an input format specification. In that case, the characters punched (or blanks) in those columns of the card so specified are then stored in the format specification replacing those appearing originally. Thus, if a routine contained the following statements:

```
VECTØR VALUES DATE = $S4,13HTØDAY IS WHAT*$
READ FØRMAT DATE
PRINT FØRMAT DATE
```

and the card read was

ΔΔΔΔ4 JULY 1776ΔΔ

the characters TØDAY IS WHAT would be replaced in the format specification by 4 JULY 1776ΔΔ which would then be printed with the PRINT FØRMAT. The output would use the first character of the S4 field (a blank) for carriage control and the printout would thus be three blanks, followed by "4 JULY 1776."

The tabulation (T) specification is used to control the column in which the next field is to start. As such, it is not a field specification, although it has the same form as one. Normally, the first field of a line specification begins in column 1 and the successive fields begin in the columns which follow the last column used by the previous field. Tabulation in formatting is quite similar to tabulation on a typewriter. In another sense, a T specification is similar to an S specification except that the column count for a T specification begins in the first column of the card or paper, rather than after the last field as in an S specification. This specification, which is particularly useful to center output on a page, is formally given by the

Definition of a T specification:

<T-spec> is T { <deci-con> or <format-var-use>#}

The value following the T specifies the column in which the next field is to begin. For example, the following two specifications produce idential results:

```
S4, 4HI = , I6, S10, E12.4
T5, 4HI = , I6, T25, E12.4
```

If consecutive fields are to be described by the same field specification, repetition may be avoided by prefixing the field specification by its multiplicity. Thus, C1, C1, C1, C1 and 4C1 each specify four consecutive fields of one character each. Similarly, repetition of a group of field specifications can be stated by enclosing the group in parentheses and preceding the left parentheses by the multiplicity. Thus, two equivalent specifications are 3(C4, I6) and C4, I6, C4, I6, C4, I6. Multiplicity of zero (useful with format variables, discussed below) causes the field (or fields) to be skipped without any effect on that line. To formalize these

multiple descriptions and the field specification which was used in the definition of format specifications in the previous section, we now give the Definition of multiple field specification and field specification:

<mult-field-spec> is <mult> <basic-field-spec> or <mod-field-spec>#

<field-spec> is <mult-field-spec> or
<mult> ( L{ <field-spec> } )

Some examples of field specification are as follows:

```
4(S8,4HXΔ=Δ,E15.6)
4(I3,F15.8)
I2,2(I3,4(S1,C2,I5))
```

The modified field specifications which allow certain flexibility in the use of fields are discussed in Section E.5.

## E.4 Operation of Formatted Input and Output

Now that the form of the input and output lists has been presented and the various fields which may be used in format specifications have been discussed, these can be combined to consider in detail the operation of input and output statements under format control. As mentioned before, each item on an input or an output list is associated with a field in the format specification and these must be compatible in mode. The input or output operation continues until the list is exhausted or an error is encountered. If not all of the format specification has been used when the list is exhausted, no error results. This is, in fact, considered to be a normal way of using formats. For example, if the data always consisted of fourteen or fewer integers, each having at most five characters including sign, it would be quite natural to use a format specification of "I2,14I5" to read this data. Here the I2 field is used for a count of the number of data items. The input statement

```
READ FØRMAT $I2,14I5*$, I, N(1) ... N(I)
```

would then read the desired number of integers from the data card and ignore the rest of the card. Another input statement following this one would begin reading data from the next card.

If the end of the format specification is reached before the input or output list being used is exhausted, part or all of the format specification is reused until the list is exhausted or an error results. In the most simple case, where the format specification contains no parentheses (other than in H fields or format variables), the entire specification is used again with a new card or a new line

being used. Thus, if one attempted to read more than fourteen integers with the above statement (i.e. I > 14), the fifteenth element of N would be read from the first two columns of the next card and the next fourteen elements from 5 column fields. This is clearly not a practical solution, since it is unlikely that the fifteenth element could be counted on to be smaller than the others.

If a format specification contains parentheses (other than in H fields and format variables), and the input or output list being used is not exhausted when the end of the specification is reached, the part of the specification to the right of the last (rightmost) unnested left parenthesis, along with the multiplicity preceding that left parenthesis, if any, is used. If this multiplicity is one, then a new card or line is started; however, if this multiplicty is greater than one input or output is continued on the same line. In almost all cases the multiplicity on a portion of a format specification which is to be repeated should be one. Thus, in the above example, the specification might be changed to "I2,14I5/(S2,14I5)". In this case, on the second and all successive cards needed to satisfy the input list, the first two columns would be ignored and up to fourteen integers read from five column fields. In general, if a large number of items are to be read, the count will be placed on a separate card, and all cards after that will have the same format. Thus, allowing up to 9999 input items in the above example, the format specification "I4/(14I5)" might be used. This would read only the count from the first field, with the data occupying fourteen five column fields in columns 1 to 70 of the succeeding cards. As another example of this, the statement

```
PRINT FØRMAT $I6/(4F14.4)*$, N, A ... A(18)
```

would print an integer N in a 6 column field on the first line, and four floating point values from the vector A on each succeeding line (except the last which would have only the two remaining values).

Input and output blocks indicate that a set of items are to be read or written. These items are usually all from the same array and just how many items are indicated by the block depends on the structure of that array. For example, A(1,1) ... A(3,4) involves 12 items if the array A has 4 elements per row, but 24 elements if each row has 10 elements. The block heading which is automatically supplied with the PRINT RESULTS statement is not provided with formatted output. Furthermore, an array is not automatically printed out by rows as with PRINT RESULTS. In fact, any such automatic features would be in direct opposition to the philosophy of formatting; where the user must have complete control of what is being done.

The iteration element in input and output lists is quite powerful. Essentially it allows the full increment form of the iteration control statement to be included in the input or output list in an abbreviated form. For example, the input statement used earlier in this section with block notation can be written as fol lows:

```
READ FØRMAT  $I2,1415*$, I, (J=1,1,J.G.I, N(J))
```

The format input or output list controlled by the iteration may be of any complexity (including blocks, nested iteration elements and even expressions in the case of output lists). For example, if A is an integer array and I and J are integers which are declared by

```
INTEGER I, J, A(3*4)
```

then the following three input statements are equivalent:

```
READ FØRMAT $4I3*$, A(1,1) ... A(3,4)

READ FØRMAT $4I3*$, (I=1,1,I.G.3, A(I,1) ... A(I,4))

READ FØRMAT $4I3*$, (I=1,1,I.G.3, (J=1,1,J.G.4, A(I,J)))
```

The first statement is actually the most simple to use and also the most efficient, but it is not as flexible. For example, with the iteration form one could rearrange the elements of a row in any order as in the following input statement

```
READ FØRMAT $4I3*$, (I=1,1,I.G.3, A(I,2),A(I,3),A(I,1),A(I,4))
```

which would permute the fields from the card upon reading them into the array. Similarly, the output statement

```
PRINT FØRMAT $4I5*$, (I=0,2,I.G.N, I*A(I))
```

would print the value of the product I*A(I) for only the even I's.

In order to print the names and values of the integer mode variable, N, and a floating point mode variable, X, in the same style as would be produced by the statement

```
PRINT RESULTS N, X
```

the following declaration and print statement might be used

```
VECTØR VALUES F1=$1HO,S15,H+N =+,I14,1H,,S15,H+X =+,F14.6*$

PRINT FØRMAT F1, N, X
```

These statements, however, are only valid for printing the variable N and X. It should also be noted that this statement will not automatically choose an E or an F field specification depending on the size of the value being printed (as the PRINT RESULT statement does). If the value is too large for the F field specification an error will result. If it is too small, zero or only a few significant digits may be printed. The format could be modified so as to print any variable name as follows:

```
VECTØR VALUES F2=$1HO,S10,C6,2H =,I14,1H,,S10,C6,2H =,
1E14.6*$
```

and then the output statement could be of the form

```
PRINT FØRMAT F2, NAME(0), N, NAME(1), X
```

where NAME(0) and NAME(1) contain, right justified, the BCD names of the variables N and X, respectively. As an exercise, the reader might modify this specification to allow subscripts on the variable names. Actually, the form of output provided by PRINT RESULTS is fairly complex to produce and in many cases is the most convenient one to use. Although the use of formats gives the user almost complete flexibility in the form his input and output may take, he must invest considerable effort in the specification of formats and their coordination with his input and output lists.

A format specification that describes four output lines for the title of a table and headings for its columns is included in the following output statement:

```
PRINT FØRMAT $T58,H+TITLE FOR A TABLE+//S51,1HX,S15,1HY,S15,
11HZ/*$
```

The first line is centered using the tabulation specification. Since a null line is specified between the two consecutive slashes, a line is skipped and then the headings for three columns are printed. Finally, a line is skipped after these headings, because of the null line specified between the last slash and the asterisk. Note that an output list is not required with this statement. A format specification that could be used to print a line of this table, where each item in the table is a floating point mode number in F format (with two decimal places), could be preset with the following declaration:

```
VECTØR VALUES F3 =$S48, 3(F7.2, S9)*$
```

The rounding performed on a number X = 27.139546 under various formats could be demonstrated by the output statement

```
      PRINT FØRMAT $S10,F10.6,F10.5,F10.4,F10.3,F10.2,F10.1,F10*$,
     1     X, X, X, X, X, X, X
```

which would print the following line:

```
27.139546  27.13955   27.1395    27.140     27.14      27.1        27
```

An input statement containing a format specification which would read in 30 letter grades from columns 2, 4, 6, ... of a card and place the letters in the left-most position in the variables is as follows:

```
      READ FØRMAT $30(S1,C1)*$, G(1) ... G(30)
```

If more or less than 30 grades were to be read, such card(s) might be preceded by a card containing the count N of the grades. Then the following input statement would read the N grades:

```
      READ FØRMAT $I4/(30(S1,C1))*$, N, G(1) ... G(N)
```

Finally, consider the problem of printing just the lower triangular part of a square array A of integers. Assuming that the integers are all less than 1000 in magnitude, a five column field will allow for a sign and at least one space on the left. Assume also that it is desired to print these values double spaced on a separate page with column and row numbers. Under these conditions it is possible to handle at most a 25 × 25 array. The following MAD statements will print the desired table where N is the dimension of A.

```
      PRINT FØRMAT $6H1    A,25I5*$, (J=1,1,J.G.N,J)
      THRØUGH S, FØR I = 1, 1, I .G. N
S     PRINT FØRMAT $1H0,26I5*$, I, (J=1,1,J.G.I, A(I,J))
```

In the case of N = 5, the printing of A would appear as follows:

```
A    1    2    3    4    5

1  -31

2   27   46

3   16  162    5

4  -94   -0   49  -13

5   21  127   11    4   43
```

## E.5 Field Modification

In most cases, the basic field specification can be modified to obtain some special effects. The modification specifications precede the basic field specifications and, if more than one is used, they may appear in any order. In this Appendix we will describe only the more common modifiers that are available.

The base (B) modification may be applied only to integer (I) fields and causes a specified base other than 10 (decimal) to be used. This base is specified by an integer preceding the letter "B". If a base greater than 10 is used, additional characters are taken from the beginning of the alphabet; however, bases larger than 19 are not allowed. Note that the sign bit is interpreted as a sign and also that leading zeros are suppressed. Thus, K12 and 8BI12 are not quite the same. As an example, the integer 22 would be printed as 22 using an I2 specification, as 1A using the specification 12BI2 and as 10110 using 2BI5. Formally, we have the Definition of B modifier and modified I field specification:

<B-mod> is { <deci-con> or <format-var-use>$^{\#}$ } B

<mod-I-field-spec> is <B-mod> <mult> <I-field-spec>

The left justification (L) modification may be applied only to octal (K) fields and causes numbers to be left justified on input, instead of the normal right justification. If the modifier 'L' precedes an octal (K) field specification, the first digit is made non-zero, with trailing zeros added. Thus, 754 read with an LK3 specification is stored as 754000000000 rather than as 000000000754, which would be the case with a K3 specification. Formally, we have the
Definition of L modified K field specification:

<mod-K-field-spec> is L <mult> <K-field-spec>

Both double precision (D) and scale factor or power (P) modifications are available for floating point (E and F) fields, the first allowing double precision input-output and the second applying a scale factor on output only. When using the D modifier, a single field corresponds to two words in memory and both of these parts of the number must be indicated on the input or output list, with the most significant part of the number first. Internally, each half of the number has its own exponent and mantissa, and is thus suitable for use with the double precision subroutines. In the scale factor modification, an integer (positive or negative) preceding the letter P has quite different meanings for F fields and E fields. For F fields, the number printed is the value stored, multiplied by 10 to the specified integer power.

Thus, the number 7.634 would be printed as 763.400 using the specification 2PF8.3 or as .076 using -2PF8.3. For E fields, only the form of the number output is changed, not its value. The 7.634 would be printed as .7634E+01 using the format E10.4, as 7.6340E+00 using 1PE10.4, or as 7634.00E-03 using 4PE11.2. (Note 4PE10.4 would not be large enough and would result in an error). Formally, we give the

Definition D and P modifiers, and modified E and F field specifications:

<P-mod> is { { <null> or -} <deci-con> or <format-var-use>$^{\#}$ } P

<D-P-mod> is D or <P-mod> or D <P-mod> or <P-mod> D

<mod-E-field-spec> is <D-P-mod> <mult> <E-field-spec>

<mod-F-field-spec> is <D-P-mod> <mult> <F-field-spec>

Both right justification (R) and zero completion (Z) modifications are available for character (C) fields, the first causing right justification of data read from character fields, and the second replacing trailing or leading blanks with zeros. Characters brought in with a C field of size less than 6, e.g. C2, are normally left justified. If the R modifier is used, the word is right justified, i.e. blanks are added on the left. Thus, AB would be stored as AB$\Delta\Delta\Delta\Delta$ using C2, but as $\Delta\Delta\Delta\Delta$AB using RC2. However, A$\Delta$ would be stored as $\Delta\Delta\Delta\Delta\Delta$A$\Delta$ using RC2, i.e. blanks within the two characters are not affected. The Z modifier causes trailing or leading (with R modifier) blanks in C fields to be replaced by zeros on input. As with the R modifier, blanks within the field length are not changed. Thus A$\Delta$ would be stored as A$\Delta$0000 using ZC2 or as 0000A$\Delta$ using either RZC2 or ZRC2. Formally, this is given by the

Definition of R and/or Z modified C field specifications:

<mod-C-field-spec> is { R or Z or RZ or ZR } <mult> <C-field-spec>

Collecting the various possibilities of modified field specification into a single metalinguistic variable for use in the definition of multiple field specification given at the end of Section E.3, we have the

Definition of a modified field specification:

<mod-field-spec> is <mod-I-field-spec> or <mod-K-field-spec>
or <mod-E-field-spec> or <mod-F-field-spec>
or <mod-C-field-spec>

## E.6 Format Variables

A format variable may be used to replace any integer in a format specification (e.g. multiplicity, field width, decimal place indicator, or modifier). Any integer, floating point or Boolean variable (other than formal parameters of subroutines) in a routine may be used as a format variable, but such use must have been indicated in a format variable declaration of the form:

FØRMAT VARIABLE L{ <var-name> }

If the mode of the variable is other than normal mode, this mode must also be declared. When interpreted as a format variable, floating point mode numbers are truncated, whereas a Boolean mode 0B is interpreted as 0 and 1B as 1. The value of a format variable is established by using that variable elsewhere in the routine or by reading it in as data. When used in a format specification, format variables are enclosed in primes (e.g. 'A'), but elsewhere in the routine they appear normally without the primes. A format variable may be used in a routine, outside a format specification, in the same way as any other variable.

Definition of a format variable, format subscript and format variable use:

```
      <format-var>   is   <int-var> or <f-p-var> or <bool-var>
      <format-sub>   is   <format-var> or <int-con>
<format-var-use>     is   '<format-var-name>' or
                          '<format-var-name>(<format-sub>)' or
                          '<format-var-name>(<format-sub>, <format-sub>)'
```

Note that format variables may not be formal parameters of subroutines, and they must exist and use storage in the routine in which the format specification appears.

The following routine illustrates the use of a format variable to print out a list of the positive constants from a set of 20 integer mode constants read as data. The sum of the constants should also be printed on the same line.

```
                NØRMAL MØDE IS INTEGER
                FØRMAT VARIABLE N
                DIMENSIØN X(19), Y(19)
                READ FØRMAT $10I4*$, X(0) ... X(19)
                SUM = 0
                N = 0
                THRØUGH S, FØR I = 0, 1, I.G.19
                    SUM = SUM + X(I)
                    WHENEVER X(I) .G. 0
                        Y(N) = X(I)
                        N = N + 1
          S         END ØF CØNDITIØNAL
                PRINT FØRMAT $S1,'N'I5,S6,H+SUM = +,I6*$, Y...Y(N-1), SUM
                END ØF PRØGRAM
```

The twenty input integers are up to four characters in length and use the first 40 columns of each of two cards. The output format uses a blank for carriage control (S1). The N positive constants are then printed using an I5 field for each. This provides at least one space between constants, since all constants are at most 4 digits in length. After skipping six spaces the characters SUMΔ=Δ and the value of SUM are printed. (Note that + is used as the delimiter in the Hollerith field). If it were desired to always place the sum in the same column, then the output format specification could be

```
S1,'N'I5,T120,H+SUM = +,I6
```

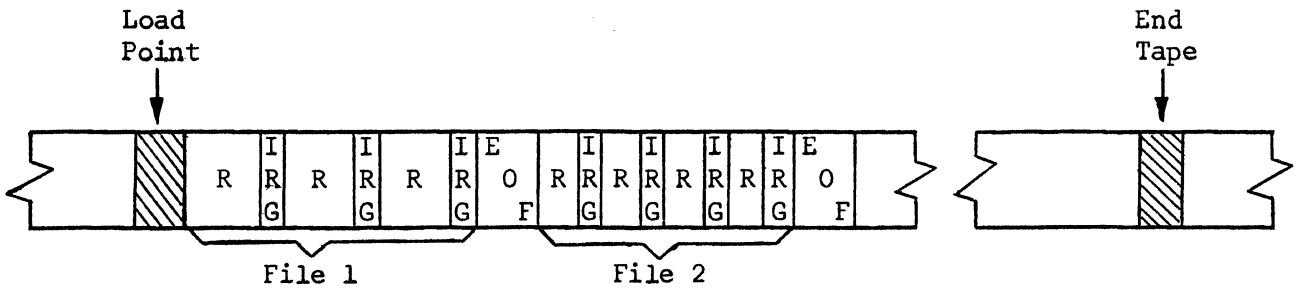
If the value of a format variable is zero when it is used as a multiplicity in a format, then the field or fields to which this multiplicity applies are completely omitted from the specification. This means that a Boolean format variable may conveniently be used to modify the format specification, determining whether certain portions of it are used or not on the basis of the true or false value of the variable. Care must be taken, however, that the input or output list used with such a format is compatible with it in all cases.

In other cases, a format variable may be used to determine the number of decimal places printed, the number base (radix) in which the results are printed and other factors in a format specification as indicated in the formal definitions.

## E.7 Input and Output on Magnetic Tape

The basic input-output statements discussed in Chapter 3 and most of the statements discussed in Section 1 of this Appendix (except for PRINT ØN LINE FØRMAT) actually read or write data on magnetic tapes; however, these tapes are the standard input or output tapes which are used for loading or punching cards, and printing listings and results of programs. In dealing with large scale problems, it is often necessary to use magnetic tapes for various other purposes, such as intermediate storage of data, preparation of data for use on other equipment, or input of data previously stored on tape. The tape units in the computer are assigned logical numbers, which are integers, by which they may be designated. These numbers vary from installation to installation and one should check with the installation to find out which numbers should be used for various purposes. Tape number 5 is usually the standard input tape and tape number 6 is usually the standard printing or listing tape.

Before proceeding with a discussion of the input-output statements and other tape manipulation statements, a brief discussion of the characteristics of magnetic tape is given. The magnetic tape described here is 1/2 inch wide and usually some 2400 feet of this tape is mounted on a reel which is 10½ inches in diameter. Two methods of recording data are possible on this tape. The first places a single BCD character with an even parity bit across the width of the tape in a frame. The second places six binary bits with odd parity in such a frame. Only whole computer words may be transmitted to and from tape and so there are normally six of these frames together corresponding to a 36 bit word inside the machine. There are three possible densities which may be used, namely, 200, 556 and 800 characters per inch. Data is structured on the tape into records and files. A record consists of a set of adjacent frames containing characters or binary bits which is separated from other data by inter-record gaps (IRG) which are actually 3/4" strips of tape on which no data is stored. A file consists of a set of adjacent records which is separated from other files by end of file (EOF) marks which are actually 3" strips of tape on which no data is stored. There is a mark on the tape at the point, called the load point, where recording may begin and also a mark at the end of the available recording area, called the end tape mark. These characteristics of tape are illustrated below:




An input and an output statement are available in MAD which function essentially the same as READ FØRMAT and PRINT FØRMAT or PUNCH FØRMAT except that they allow the tape being used to be specified by an integer mode expression. These BCD tape input and output statements are of the following forms:

READ BCD TAPE <int-exp>, <format-exp>, <format-input-list>

WRITE BCD TAPE <int-exp>, <format-exp>, <format-output-list>

The first statement reads the items on the input list from the tape designated by the value of the integer expression according to the given format specification.

The data must have been stored on tape in BCD and so must consist of legal BCD characters. The second writes from the items on the output list on the tape designated by the value of the integer expression according to the given format specification. The data being written must be legal BCD data when converted according to the format. If tapes 5 and 6 are the standard input and print tapes, respectively, then

READ BCD TAPE 5, F1, A, B, C(1) ... C(8)

would be equivalent to

READ FØRMAT F1, A, B, C(1) ... (8)

and

WRITE BCD TAPE 6, F2, P, Q, R ... R(41)

would be equivalent to

PRINT FØRMAT F2, P, Q, R, ... R(41)

When tapes are to be used for temporary storage and in some other cases, it is desirable to avoid the conversions which must take place between internal storage and formatted BCD tape. These conversions require sufficient computer time that they may introduce considerable unnecessary expense to some processes. The binary method of recording on tape does not require this complex conversion and is therefore useful in these instances. Since no conversion takes place, formats are unnecessary in the binary tape statements which follow:

READ BINARY TAPE <int-exp>, L{ <bin-input-element> }

WRITE BINARY TAPE <int-exp>, L{ <bin-output-element> }

where we have the

Definition of binary input and output elements:

<bin-input-element> is <input-var> or
<input-var> ... <input-var> or
<input-var> ,..., <input-var>

<bin-output-element> is <output-var> or <output-exp> or
<output-block>

As a result of the first statement, items are read from the designated tape until the list is exhausted. If there are more elements on the list than words in the tape record being read, reading continues into the next record. The second statement writes a single binary record containing the values of all items on the output list. In both cases the integer expression specifies the logical tape to be used. Note that iteration elements may not be used with binary tape statements since their operation cannot be carried out in the time available. Otherwise the input and output lists are the same as for BCD tape statements.

In working with tape units it is often necessary to cause various specific operations to be performed. Some of these involve the backspacing of the tape by a record or a file. These operations are called for in MAD by the following backspace tape statements:

```
BACKSPACE RECØRD ØF TAPE  <int-exp>

BACKSPACE RECØRD ØF TAPE  <int-exp> ,
          IF LØAD PØINT TRANSFER TØ  <label-var>


BACKSPACE FILE ØF TAPE  <int-exp>

BACKSPACE FILE ØF TAPE  <int-exp> ,
          IF LØAD PØINT TRANSFER TØ  <label-var>
```

The first two statements cause an attempt to be made to backspace the tape designated by the integer expression by one record. If the load point on the tape is encountered and the phrase IF LØAD PØINT TRANSFER TØ is included, then control is transferred to the statement specified by the label variable. If this phrase is not included, then no recognition is made that the load point has been encountered. The last two statements behave the same as the previous two except that they attempt to backspace the tape by a file rather than a record.

Some other tape operations involve the writing of an end of file mark, the rewinding of a tape, or the rewinding and unloading of a tape. The additional tape control statements which cause these operations to be performed follow:

```
END ØF FILE TAPE  <int-exp>

REWIND TAPE  <int-exp>

UNLØAD TAPE  <int-exp>
```

In each case the value of the integer expression designates which logical tape is involved.

Finally, it is sometimes necessary to designate the recording density which is to be used in the tape reading or writing operations which are to follow. The MAD tape density control statements are as follows:

```
SET LØW DENSITY TAPE  <int-exp>
SET HIGH DENSITY TAPE  <int-exp>
```

## E.8 Input and Output Errors

Errors involving input and output are detected only at execution time. Normally, a description of the error is printed and execution of the user's program is terminated. Such errors may be caused by a faulty format specification, incorrect calculations which produce results outside the expected range, improperly punched data cards and various other causes. The messages printed as a result of input or output errors are probably the most startling ones the user of MAD will encounter. Although they may appear to be inscrutable at first, on careful study they contain all the information one usually needs to detect the error. For example, if one attempts to read a card with an E15.5 specification and the first 15 columns of that card contain "-46.3+3Δ+29.7-4", the following error message is printed:

```
###### ILLEGAL CHARACTER ON CARD IN COLUMN  12
###### OFFENDING CHARACTER IS      '.'  (OCTAL 33)
###### ERROR FOUND WHILE PROCESSING FORMAT WORD 'E15.5*'
###### WHICH OCCURRED AT LOCATION    10005
###### PRESENT  INPUT CARD IMAGE IS
######-46.3+3 +29.7-4
###### I/O STATEMENT BEGINS
      AT LOCATION 10023
```

The first line of this message draws attention to column 12 which contains a decimal point as indicated in the second line of the message. The third line identifies the pertinent part of the format specification being used and the sixth line gives the image of the card being read. With a small amount of study one quickly realizes that the E15.5 field being read contains two floating point numbers with exponents, not one. Hence, the extra decimal point which is illegal under this format specification. An example of the error message which is printed when one attempts to print the integer 12345678 with an I6 field specification follows:

```
###### NUMBER EXCEEDS SPECIFIED FIELD WIDTH.    NUMBER IS   000057060516  (OCTAL)
###### SAME NUMBER, CONVERTED ACCORDING TO FORMAT, IS
######12345678
###### THE NUMBER CAME FROM LOCATION 10005 VIA THE STR AT LOCATION 10012
###### ERROR FOUND WHILE PROCESSING FORMAT WORD 'S2,I6*'
###### WHICH OCCURRED AT LOCATION    10003
###### PRESENT OUTPUT LINE IMAGE IS
######
###### I/O STATEMENT BEGINS
      AT LOCATION 10010
```

This error message is also clear after one studies it carefully. The locations given in these error messages are usually not needed in simple programs, but can be used in conjunction with the object program to identify the particular input or output statement and the particular format specification being used.

The error return to system which normally occurs when an input or an output error is detected can be replaced by an error return to the user's program. This is accomplished by a call to the library subroutine SETERR. a description of which follows:

SETERR. ( <label>, <int-var>, <int-var> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | SETERR.(S), SETERR.(0), SETERR.(S,E), SETERR.(S,E,T) |
| Error condition: | None |
| Action: | If an error is encountered during input or output, the error message is printed and control is transferred to the statement labeled S. If SETERR.(0) is called, format errors return control to the system, as before any calls to SETERR. The label (or zero) in the last call for SETERR. that was executed before the error is the one used. If E is included, the error number is stored in E. If T is included, the logical number of the tape being used when the error occurred is stored in T. |

The system automatically inserts an end-of-file mark after the last set of input data for each program on the input tape. If the end-of-file mark is encountered when an input instruction is executed which involves the standard input tape, the comment

****ALL INPUT DATA HAVE BEEN PRØCESSED

is normally printed and the execution of the program is terminated. If an end-of-file on other tape is encountered the message is:

****END ØF FILE ØN SCRATCH TAPE

This procedure can be modified by executing the library subroutine SETEØF. before the input statement is executed. The description of SETEØF. is as follows:

SETEØF. ( <label>, <int-var> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | SETEØF.(S), SETEØF.(0), SETEØF.(S,T) |
| Error condition: | None |
| Action: | If SETEØF.(S) has been executed and an end-of-file is encountered, control is transferred to the statement labeled S and no message is printed. If SETEØF.(0) is executed, an end-of-file causes normal termination of the program and printing of the message. The last call for SETEØF. prior to encountering the end-of-file is the one in effect. If T is included, the logical number of the tape being used when the EOF was encountered is stored in T. |

When an end tape mark is encountered on the standard output tape, that tape is terminated with an end-of-file mark and the operator is instructed to replace that tape with a blank tape. This is also the normal procedure for other tapes; however, the user may wish to alter this action. This can be accomplished, for tapes other than the standard output tape, by executing the library subroutine SETETT. before the output statement is executed. This library subroutine is described as follows:

SETETT. ( <label>, <int-var> )

| | |
|---|---|
| Subroutine class: | Procedure |
| Typical calls: | SETETT.(S), SETETT.(0), SETETT.(S,T) |
| Error condition: | None |
| Action: | If SETETT.(S) has been executed and an end-tape mark is encountered on output, control is transferred to S and no message is printed. The execution of SETETT.(0) returns the action to the normal one of termination and printing the error message. The last call for SETETT. which has been executed prior to encountering the end-tape mark is the one in effect. If T is included, the logical number of the tape on which the end-tape mark was found is stored in T. |

Appendix F

# OTHER FEATURES OF THE MAD LANGUAGE

In this appendix, we will present some additional statements that give the MAD language great versatility. We will not, however, discuss the "Define" facility[1] which allows the user to define operators and modes (by writing assembly language programs). With this feature, a programmer can augment the language to meet his needs. A few other advanced statements have also been omitted.

## F.1 Some Additional Declarations

The equivalence declaration has the general form:

EQUIVALENCE L{( <eq-var>, <eq-var> )}

where <eq-var> is <var-name> or <vector-base>

The declaration specifies that each pair of variables enclosed in parentheses are to be stored in the same location in memory. For example,

```
EQUIVALENCE (I, J), (B, A(2))
```

specifies that I and J are to be stored in the same location and that B and A(2) are to be stored in the same location. In the case of vectors, the equivalence implies the same storage for other elements of the vector (e.g. in the example B(1) and A(3) would also be stored in the same location). It should be noted that the above declaration would have the same effect as

```
EQUIVALENCE (B(3), A(5)), (J, I)
```

Equivalent elements need not be of the same mode and, in particular, this is a way to interpret the same number in two different modes. It also allows one to use two different names for the same variable. This is useful in handling arrays, for example, where one might want to set the structure using the declarations

```
DIMENSIØN A(399,ASV)
VECTØR VALUES ASV = 2,0,20
```

If an m x n array is read in, along with the values of m and n, the correct

[1] B. Arden, B. Galler, R. Graham, The Michigan Algorithm Decoder, University of Michigan, 1965.

structure could be obtained by setting

ASV(2) = N

after the input statement. The same effect could be obtained without the use of that assignment statement by declaring

EQUIVALENCE (N, ASV(2))

Like other declarations, EQUIVALENCE may appear anywhere in the routine. Conflicting EQUIVALENCE statements (i.e. declaring the same variable equivalent to two different elements of the same vector) should be avoided.

The ERASABLE declaration assigns the variables and arrays so declared to a special section of memory. All names declared in a single declaration are assigned storage so as not to overlap, i.e. each element has its own storage location. However, variables and arrays assigned to the erasable section of memory in other erasable declarations (either in that routine or in other routines in the same program) are assigned the same memory locations. The declaration is of the form

ERASABLE L{ <id> }

or, if it is wished to include dimension information for some or all of the names,

ERASABLE L{ <id> or { <id> or ( L{ <id> }) } (<structure>) }

This declaration is particularly useful when large amounts of storage are required in subroutines (or segments of the same program) but are no longer needed after that subroutine (or segment) is completed. Formal parameters may not be declared in an ERASABLE declaration. Also variables declared to be in ERASABLE may not be preset in a VECTØR VALUES declaration.

The PRØGRAM CØMMØN declaration causes those variables and arrays so declared to be assigned to another special section of memory, different from erasable and normal routine storage. The general form of the declaration is the same as that for ERASABLE, namely

PRØGRAM CØMMØN L{ <id> or { <id> or ( L{ <id> }) } ( <structure> ) }

The purpose of this declaration is to allow a main routine and subroutines to refer to the same variables. If, in particular, identical PRØGRAM CØMMØN statements are used in a main routine and one or more subroutines and if the same storage space for the vectors and arrays is reserved in all of these routines, then the same names will refer to the same storage location in all such routines. In general, the order of names and the amount of storage specified for each determines the location in

CØMMØN referred to. Thus, for example, if the declaration (and no other dimension declarations on A,B,C)

PRØGRAM CØMMØN A(10), B, C(10*5)

were used in a routine, the vector A would be stored in the first 11 locations in CØMMØN, B in the twelfth location and the array C in the next 51. If the declaration

PRØGRAM CØMMØN X(20), B(4*4)

appeared in another routine in the same program set, the vector X would be stored in the first 21 locations in CØMMØN (corresponding to A, B and part of C in the other routine, and B would be stored in the next 17 locations. Storage for variables in the CØMMØN section of memory is allocated in the main routine and thus all variables assigned to CØMMØN must be so declared in the main routine. Furthermore, the main routine must be the first routine in the deck. More than one PRØGRAM CØMMØN declaration may be used; variables in the second declaration are treated as if they were added to the first list. Formal parameters may not appear in a PRØGRAM CØMMØN declaration. Vectors in CØMMØN can be preset (using VECTØR VALUES) only with numerical, Boolean, or alphabetical information (not labels or function names).

## F.2 Some List processing Statements

In this section, we will discuss five statements that may be used to define and utilize a "push-down" list and are particularly useful for the definition of recursive subroutines (i.e. subroutines that call themselves). A push-down list or stack (like a stack of dishes) is a list in which the first item retrieved from the list is the last item put on the list. Thus, if the items A, B and C are put on the list in that order (A first), they will be retrieved in the order C, B and A.

Each such push-down list is stored in a vector, for which memory must be reserved. The vector that is to be used at any particular time for list processing must be specified in a set list statement of the form:

SET LIST TØ <vector-ref>

This statement specifies that all future references to a list (until the execution of another set list statement with a different vector reference) refer to that vector. The base element of the referenced vector is always used to store the number of elements currently on the list, i.e. the length of the list. The compiler does not initialize the length of lists to zero and so this initialization (usually done by presetting the base element to zero) must be provided explicitly. The other list

manipulation statements do not specify the list by name, but refer to the list currently being used. Items may be added to or removed from the list in any routine in a program set. However, the vector name is used in only one of these routines and is dimensioned in that routine. Note that if SET LIST TØ X is used in two different routines, two different lists are usually referenced, since the same name normally refers to different storage locations in different routines. If a single list were required, statements such as

```
INTEGER X(100)
VECTØR VALUES X = 0
SET LIST TØ X
```

would typically appear at the beginning of the main routine and all list manipulation statements would then refer to X and would use storage in the main routine.

Items may be added to a list using a save data statement having the form:

```
SAVE DATA  <format-output-list>
```

The items in the output list are stored in consecutive locations in the current list vector. If, for example, the list were just established as X as shown above (and no items were yet added to the list), the statement

```
SAVE DATA 3.6, 2*I, A, B(3)...B(7)
```

were executed, the value 3.6 would be stored in X(1), the value of 2*I in X(2) and so forth with B(7) stored in X(8). The element X(0) is used to store the length of the list and thus has the value zero before the statement is executed and the value 8 after execution. Mode conversion does not take place when using the list processing statements.

Items may be retrieved from lists using a restore data statement having the form:

```
RESTØRE DATA  <format-input-list>
```

An example of such a statement is

```
RESTØRE DATA B(7)...B(3), C, P, Q
```

where the last item on the list is stored in B(7), the next to last item in B(6) and so forth. Thus, if the statement were used as the next list operation statement after the SAVE DATA statement above, the elements of the B vector B(3)...B(7) would be restored to their original values, the value of C would be that of A (at the time it was saved), the value of P would be that of 2*I and Q would equal 3.6. Since no mode conversion takes place, C and A (and I and P) should be of the same mode, and Q should be floating point mode if the correct values are to be interpreted. Note,

that if Q were integer mode, it would be interpreted as the octal number 202714631463 (greater than $10^{10}$).

The save and restore data statements (as well as the set list statement) cause subroutines to be called. The subroutine used in saving data adds the items to the end of the list and increments the length of the list. In restoring data, items are transferred from the list and the length is decremented. The vector still contains the same items after a restore data statement with only the length modified to indicate that the list no longer includes the restored items. The length is treated as an integer by these subroutines. Thus, if this length is to be used elsewhere in the routine, the vector used for the list must be in integer mode.

In the compilation of a subroutine, one storage location is reserved for the address in the calling program to which control is to return when execution is completed. If a subroutine is to be able to call itself, it is necessary to store the address of the return from the first call before it can call itself and then restore that address upon return. The statements used for this purpose are

```
SAVE RETURN
RESTØRE RETURN
```

Note that this information is stored on the current list (the same list as would be used for SAVE DATA). In order to define recursive subroutines, it is also necessary to save some data (using SAVE DATA) that will be required upon returning to the routine. In order to better understand the use of these statements, let us consider an example of a program to calculate the sum of the first n integers recursively. We define

$$f(0) = 0$$

$$f(n) = f(n-1) + n$$

Although this function can easily be evaluated directly, either using a THRØUGH loop or using the relation $f(n) = \frac{n(n+1)}{2}$, it provides us with a simple example of a recursive routine. A main routine is required to read data, call the function which we will designate as SUM., and print the results. The subroutine SUM. calculates f(n) in terms of f(n-1) (for n greater than zero) by recursively calling itself. Thus, the basic calculations are included in the statements:

```
WHENEVER N.E.0, FUNCTIØN RETURN 0
T = SUM. (N-1)
FUNCTIØN RETURN T + N
```

Thus T is calculated as the sum of the first n-1 integers and then this routine calculates and returns the sum of the n integers by adding n to the sum of the n-1 integers. Since SUM. may be called many times, it is necessary to save the return location and the value of n before calling the routine again and restore it upon

return. Thus, the complete subroutine is

```
EXTERNAL FUNCTIØN SUM.(N)
NØRMAL MØDE IS INTEGER
WHENEVER N.E.0, FUNCTIØN RETURN 0
SAVE RETURN
SAVE DATA N
T = SUM.(N)
RESTØRE DATA N
RESTØRE RETURN
FUNCTIØN RETURN T + N
END ØF FUNCTIØN
```

A main routine could be

```
        NØRMAL MØDE IS INTEGER
        DIMENSIØN X(100)
        VECTØR VALUES X = 0
        SET LIST TØ X
START   READ AND PRINT DATA     N
        PRINT RESULTS N, SUM.(N)
        TRANSFER TØ START
        END ØF PRØGRAM
```

The dimension statement allows calculations for N $\leq$ 50 since two items are stored on the list for each recursive call for SUM.. In order to understand the operation of this program, let us consider the order of execution for N = 3.

```
SUM.(3)
3 .E. 0          False
Save Return      (to main)
Save 3
SUM.(2)
2 .E. 0          False
Save Return      (to SUM.)
Save 2
SUM.(1)
1 .E. 0          False
Save Return      (to SUM.)
Save 1
SUM.(0)
0 .E. 0          True
Return 0
T = 0
Restore N = 1
Restore Return (to SUM.)
Return 0 + 1   (to SUM.)
T = 1
Restore N = 2
Restore Return (to SUM.)
Return 1 + 2   (to SUM.)
T = 3
Restore N = 3
Restore Return (to main)
Return 3 + 3   (to main)
Print      N = 3, ... = 6
```

# Appendix G

## SYNTACTIC SUMMARY OF THE MAD LANGUAGE

This appendix contains a summary of the syntax of the MAD Language. The statements of MAD, organized by type, are presented first. Next, the operators of the MAD Language are given. Following this, the metavariables used in the text and their definitions are listed in alphabetic order. The numbers on the right refer to the pages on which these statements, operators, and metavariables are introduced and discussed.

### G.1 Declarations

| | | Page |
|---|---|---|
| <mode-name> | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 97 |
| BØØLEAN | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 97, 156 |
| FLØATING PØINT | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 97 |
| FUNCTIØN NAME | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 97, 264 |
| INTEGER | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 97 |
| STATEMENT LABEL | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 97, 147 |
| NØRMAL MØDE IS | <mode-name> | 97 |
| DIMENSIØN | L{ {<id> or (L{<id>})} (<structure>) } | 109 |
| VECTØR VALUES | <vector-base> = L{ <constant> or <char-string> } | 112 |
| VECTØR VALUES | <vector-block> = <constant> | 114 |
| FØRMAT VARIABLE | L{ <var-name> } | 395 |
| EQUIVALENCE | L{( <eq-var>, <eq-var> )} | 403 |
| ERASABLE | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 404 |
| PRØGRAM CØMMØN | L{ <id> or { <id> or (L{<id>})} (<structure>) } | 404 |

## G.2 Input-Output Statements


| | Page |
|---|---|
| READ DATA | 116 |
| READ AND PRINT DATA | 116 |
| PRINT RESULTS L{ <output-element> } | 120 |
| PRINT BCD RESULTS L{ <output-element> } | 120 |
| PRINT ØCTAL RESULTS L{ <output-element> } | 120 |
| PRINT CØMMENT {$ <c-c-c> $S_0^{131}$ { { <character> not $ } or $$ } $ or $$} | 121 |
| READ FØRMAT <format-exp>, <format-input-list> | 377 |
| LØØK AT FØRMAT <format-exp>, <format-input-list> | 377 |
| PRINT FØRMAT <format-exp>, <format-output-list> | 378 |
| PRINT ØN LINE FØRMAT <format-exp>, <format-output-list> | 378 |
| PUNCH FØRMAT <format-exp>, <format-output-list> | 378 |
| READ BCD TAPE <int-exp>, <format-exp>, <format-input-list> | 397 |
| WRITE BCD TAPE <int-exp>, <format-exp>, <format-output-list> | 397 |
| READ BINARY TAPE <int-exp>, L{ <bin-input-element> } | 398 |
| WRITE BINARY TAPE <int-exp>, L{ <bin-output-element> } | 398 |

## G.3 Substitution or Assignment Statements

| | | | Page |
|---|---|---|---|
| <a-var> | = | <a-exp> | 139 |
| <label-var> | = | <label-exp> | 147 |
| <bool-var> | = | <bool-exp> | 171 |
| <subr-var> | = | <subr-exp> | 265 |

## G.4 Transfer and Special Statements

| | |
|---|---|
| TRANSFER TØ <label> | 148 |
| TRANSFER TØ <label-name> ( <a-exp> ) | 152 |
| CØNTINUE | 147 |
| END ØF PRØGRAM | 95 |

## G.5 Conditional Statements

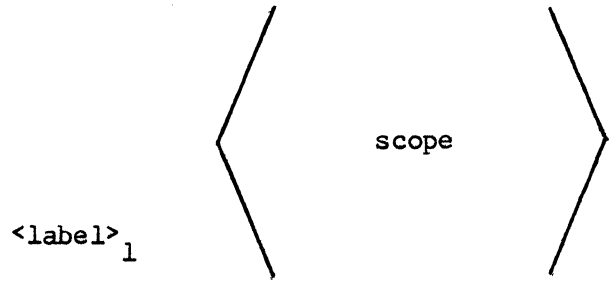
| | |
|---|---|
| WHENEVER <bool-exp>, <simple-statement> | 160 |
| WHENEVER $<\text{bool-exp}>_1$<br>$<\text{statement-set}>_1$<br>ØR WHENEVER $<\text{bool-exp}>_2$<br>$<\text{statement-set}>_2$<br>•<br>•<br>•<br>•<br>•<br>ØTHERWISE<br>$<\text{statement-set}>_0$<br>END ØF CØNDITIØNAL | 173 |

## G.6 Iteration Statements

Page

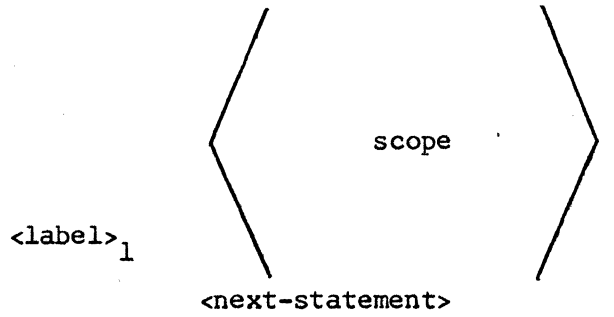
<initialization> 200

THRØUGH <label>$_1$, FØR <v> = <i>, <s>, <terminate>




<initialization> 205

THRØUGH <label>$_1$, FØR VALUES ØF <it-var> = L{ <value> }



## G.7 Subroutine Statements


| | Page |
|---|---|
| <proc-call> | 266 |
| EXECUTE <proc-call> | 266 |
| EXTERNAL FUNCTIØN <subr-con> ( L{ <formal-par> } )<br>EXTERNAL FUNCTIØN ( L{ <formal-par> } )<br>EXTERNAL FUNCTIØN <subr-con><br>EXTERNAL FUNCTIØN | 279 |
| ENTRY TØ <subr-con> | 281 |
| FUNCTIØN RETURN<br>FUNCTIØN RETURN <exp> | 282 |
| ERRØR RETURN | 282 |
| END ØF FUNCTIØN | 283 |
| INTERNAL FUNCTIØN <subr-con> ( L{ <formal-par> } )<br>INTERNAL FUNCTIØN ( L{ <formal-par> } )<br>INTERNAL FUNCTIØN <subr-con><br>INTERNAL FUNCTIØN | 303 |
| INTERNAL FUNCTIØN <subr-con> ( L{ <formal-par> } ) = <exp> | 304 |

## G.8 List Control Statements


| | Page |
|---|---|
| SET LIST TØ <vector-ref> | 405 |
| SAVE DATA <format-output-list> | 406 |
| RESTØRE DATA <format-input-list> | 406 |
| SAVE RETURN | 407 |
| RESTØRE RETURN | 407 |


## G.9 Tape Control Statements


| | |
|---|---|
| BACKSPACE RECØRD ØF TAPE <int-exp> | 399 |
| BACKSPACE RECØRD ØF TAPE <int-exp> , IF LØAD PØINT TRANSFER TØ <label-var> | 399 |
| BACKSPACE FILE ØF TAPE <int-exp> | 399 |
| BACKSPACE FILE ØF TAPE <int-exp> , IF LØAD PØINT TRANSFER TØ <label-var> | 399 |
| END ØF FILE TAPE <int-exp> | 399 |
| REWIND TAPE <int-exp> | 399 |
| UNLØAD TAPE <int-exp> | 399 |
| SET LØW DENSITY TAPE <int-exp> | 399 |
| SET HIGH DENSITY TAPE <int-exp> | 399 |

## G.10 Operators


| | Page |
|---|---|
| + | 128, 129 |
| - | 128, 129 |
| * | 128, 129 |
| / | 128, 129 |
| .ABS. | 128 |
| .P. | 128, 129 |
| .N. | 141, 142 |
| .A. | 141, 142 |
| .V. | 141, 142 |
| .EV. | 141, 142 |
| .LS. | 142 |
| .RS. | 142 |
| .G. | 157 |
| .GE. | 157 |
| .L. | 157 |
| .LE. | 157 |
| .E. | 157 |
| .NE. | 157 |
| .NØT. | 168, 169 |
| .ØR. | 168, 169 |
| .AND. | 168, 169 |
| .EXØR. | 168, 169 |
| .EQV. | 168, 169 |
| .THEN. | 168, 169 |

## G.11 Definitions of Metavariables

Page

<a-con> 104

is <int-con> or <f-p-con>

<a-exp> 106, 132

is <a-con> or <a-var> or <a-fn-call> or <bit-word>
or (<a-exp>) or <un-a-op> <a-exp>
or <a-exp> <bin-a-op> <a-exp>

<a-fn-call> 132, 266

is a function call that returns a floating point or integer mode value

<a-var> 107

is a variable which has integer or floating point mode

<actual-par> 265

is <var> or <exp> or <label> or <subr-des> or <vector-ref>
or <array-name>

<array-structure> 109, 315

is <array-structure-1> or <array-structure-2>

<array-structure-1> 313

is L*{ <subscript-range> }

<array-structure-2> 315

is <uns-int-con> , <structure-vector>

<basic-field-spec> 380

is <S-field-spec> or <I-field-spec> or <E-field-spec>
or <F-field-spec> or <K-field-spec> or <C-field-spec>
or <H-field-spec>

<bin-a-op> 128

is + or - or * or / or .P.

Page

<bin-bit-op> 142

is .A. or .V. or .EV. or .LS. or .RS.

<bin-bool-op> 169

is .ØR. or .AND. or .EXØR. or .EQV. or .THEN.

<bin-input-element> 398

is <input-var> or <input-var> ... <input-var> or
<input-var> ,..., <input-var>

<bin-output-element> 398

is <output-var> or <output-exp> or <output-block>

<bit-word> 132, 142

is <int-exp> or <un-bit-op> <bit-word> or
<bit-word> <bin-bit-op> <bit-word>

<blank> 89

is an empty space of the same size as that usually occupied by a letter, a digit or a symbol

<B-mod> 393

is { <deci-con> or <format-var-use> } B

<bool-con> 104, 169

is <truth-con> or <uns-con> M2

<bool-exp> 160, 169

is <bool-con> or <bool-var> or <rel> or <bool-fn-call>
or ( <bool-exp> ) or <un-bool-op> <bool-exp>
or <bool-exp> <bin-bool-op> <bool-exp>

<bool-fn-call> 169, 266

is a function call that returns a Boolean mode value

<bool-var> 107, 169

is a variable which has Boolean mode

Page

<c-c-c> 121

is + or <blank> or 0 or - or 1 or 2 or 4 or 6 or 8 or 9

<C-field-spec> 382

is C <width>

<char-con> 101

is $ $S_0^6${{<character> not $} or $$}$

<char-string> 112

is $ $S_7^*${ {<character> not $} or $$} $

<character> 89

is <letter> or <digit> or <symbol> or <blank>

<constant> 104

is <a-con> or <bool-con>

<d> 386

is <H-char>$_1$

<data-item> 116

is <data-var> = L{ <constant> or <char-string> }

<data-set> 116

is L{ <data-item> } *

<data-var> 116

is <var-name> or <var-name> ( L{ <deci-con> } )

<deci-con> 101

is <digit> or <deci-con> <digit>

<digit> 89

is 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9

<D-P-mod> 394

is D or <P-mod> or D <P-mod> or <P-mod> D

Page

<E-field-spec> 384

is E <width> or E <width> • <places>

<eq-var> 403

is <var-name> or <vector-base>

<exp> 265

is <a-exp> or <bool-exp> or <label-exp> or <subr-exp>

<F-field-spec> 384

is F <width> or F <width> • <places>

<field-spec> 388

is <mult-field-spec> or <mult> ( L{ <field spec> } )

<fn-call> 266

is call for a function type of subroutine

<formal-par> 279

is <var-name> or <label-name> or <subr-var-name> .

<format-exp> 379

is <int-vector-ref> or $<format-spec>*$

<format-input-element> 377

is <input-var> or <input-var> ... <input-var> or
<input-var> ,..., <input-var> or
( <v> = <i> , <s> , <terminate> , <format-input-list> )

<format-input-list> 377

is L{ <format-input-element> } or <null>

<format-output-element> 378

is <output-var> or <output-exp> or <output-block> or
( <v> = <i> , <s> , <terminate> , <format-input-list> )

<format-output-list> 378

is L{ <format-output-element> } or <null>

Page

<format-spec> 379

is <line-spec> or <mult> ( <format-spec> )

or <format-spec> / <format-spec> or <format-spec> , <format-spec>

<format-sub> 395

is <format-var> or <int-con>

<format-var> 395

is <int-var> or <f-p-var> or <bool-var>

<format-var-name> 395

is <id>

<format-var-use> 395

is '<format-var-name>' or '<format-var-name>(<format-sub>)' or

'<format-var-name>(<format-sub>, <format-sub>)'

<f-p-con> 104

is <sign> <uns-f-p-con>

<f-p-con-1> 102

is <deci-con>. or .<deci-con> or <deci-con> . <deci-con>

<f-p-con-2> 102

is {<deci-con> or <f-p-con-1>}E<sign>$\underline{S}_1^2${<digit>}

<f-p-exp> 136

is an arithmetic expression of floating point mode

<f-p-var> 107

is a variable which has floating point mode

<H-char> 386

is { <character> not $ } or $$

<H-field-spec> 386

is <width> H $\underline{S}_w^w${ <H-char> } or

H <d> $\underline{S}_0^*${ <H-char> not <d> } <d>

Page

<i> 199

is <a-exp>

<id> 96

is <letter> $S_0^5${<letter> or <digit>}

<I-field-spec> 381

is I <width>

<initialization> 199

is a set of MAD statements, often an empty set

<input-var> 377

is <a-var> or <bool-var>

<int-con> 103

is <sign> <uns-int-con>

<int-exp> 106, 136

is an arithmetic expression of integer mode

<int-var> 107

is a variable which has integer mode

<it-var> 205

is <a-var> or <bool-var> or <label-var> or <subr-var>

<K-field-spec> 382

is K <width>

<label> 146

is <label-name> or <label-name> ( <deci-con> )

<label-exp> 147

is <label-var> or <label-fn-call>

Page

<label-name> 146

is <id>

<label-var> 146

is <label-name> or <label-name> ( <a-exp> )

<letter> 89

is A or B or C or D or E or F or G or H or I or J or K or L or M or
N or Ø or P or Q or R or S or T or U or V or W or X or Y or Z

<line-spec> 379

is L{ <field-spec> or <T-spec> } or <null>

<mod-C-field-spec> 394

is { R or Z or RZ or ZR } <mult> <C-field-spec>

<mod-E-field-spec> 394

is <D-P-mod> <mult> <E-field-spec>

<mode-ind> 103

is M <octit>

<mode-name> 97

is INTEGER or FLØATING PØINT or BØØLEAN or STATEMENT LABEL or
FUNCTIØN NAME

<mod-F-field-spec> 394

is <D-P-mod> <mult> <F-field-spec>

<mod-field-spec> 394

is <mod-I-field-spec> or <mod-K-field-spec> or <mod-E-field-spec>
or <mod-F-field-spec> or <mod-C-field-spec>

<mod-I-field-spec> 393

is <B-mod> <mult> <I-field-spec>

<mod-K-field-spec> 393

is L <mult> <K-field-spec>

Page

<mult> 379

is <deci-con> or <format-var-use> or <null>

<mult-field-spec> 388

is <mult> <basic-field-spec> or <mod-field-spec>

<next-statement> 199

is any executable statement

<null> 91

is

<octit> 101

is 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7

<octa-con> 101

is $S_1^{12}${ <octit> } K or $S_1^{n}${ <octit> } K <deci-con>

<output-block> 120, 378

is <output-var> ... <output-var> or
<output-var> ,..., <output-var>

<output-element> 120

is <output-var> or <output-exp> or <output-block>

<output-exp> 120, 378

is <a-exp> or <bool-exp>

<output-var> 120, 378

is <a-var> or <bool-var>

<places> 384

is <deci-con> or <format-var-use>

<proc-call> 266

is a call for a procedure type subroutine

<P-mod> 394

is { { <null> or -} <deci-con> or <format-var-use> } P

Page

<rel> 157

is <a-exp> <rel-op> <a-exp>

<rel-op> 157

is .G. or .GE. or .L. or .LE. or .E. or .NE.

<s> 199

is <a-exp>

<scope> 199

is a set of MAD statements, excluding only such statements as END ØF PRØGRAM. If any part of a compound statement is included, that entire compound statement must be in <scope>.

<S-field-spec> 381

is S <width>

<sign> 100

is + or <null> or -

<simple-statement> 160

is any executable statement (not a declaration) except END ØF PRØGRAM, simple and compound conditional statements, iteration statements, or the ENTRY TØ statement which is used in subroutines

<structure> 97, 109, 312

is <uns-int-con> or <array-structure>

<structure-vector> 315

is <vector-base>

<subr-call> 266

is <subr-des> or <subr-des> ( L{ <actual-par> } )

or <subr-des> ( L{ <actual-par> } , <label> )

<subr-con> 264

is <id> .

<subr-des> 264

is <subr-con> or <subr-var-name> ( <a-exp> ) .

Page

<subr-exp> 265

is <subr-con> or <subr-var> or <subr-fn-call>

<subr-var> 264

is <subr-var-name> or <subr-var-name> ( <a-exp> )

<subr-var-name> 264

is <id>

<subscript-list> 106

is <a-exp> or <a-exp> , <a-exp> or $L_3^*\{$ <int-exp> }

<subscript-range> 313

is <uns-int-con> or ( $<\text{uns-int-con}>_1$ ... $<\text{uns-int-con}>_2$ )

<symbol> 89

is + or - or * or / or , or . or = or ' or $ or ( or )

<T-spec> 387

is T { <deci-con> or <format-var-use> }

<terminate> 199

is <bool-exp>

<truth-con> 103

is 1B or 0B

<un-a-op> 128

is + or - or .ABS.

<un-bit-op> 142

is .N.

<un-bool-op> 169

is .NØT.

<uns-con> 103

is <deci-con> or <octa-con> or <char-con> or <f-p-con-1>
or <f-p-con-2> or <truth-con>

Page

<uns-f-p-con> 104

is <f-p-con-1> or <f-p-con-2> or <uns-con> M0

<uns-int-con> 103

is <deci-con> or <octa-con> or <char-con> or <uns-con> M1

<v> 199

is <a-var>

<value>

is <a-exp> or <bool-exp> or <label-exp> or <subr-exp>

<var> 106

is <var-name> or <var-name> (<subscript-list>)

<var-name> 106

is <id>

<vector-base> 111

is <vector-name> or <vector-name>(<uns-int-con>)

<vector-block> 112

is $\langle\text{vector-name}\rangle_1$ ($\langle\text{uns-int-con}\rangle_1$) ... $\langle\text{vector-name}\rangle_1$ ($\langle\text{uns-int-con}\rangle_2$)

or $\langle\text{vector-name}\rangle_1$ ... $\langle\text{vector-name}\rangle_1$ (<uns-int-con>)

<vector-name> 111

is <id>

<vector-ref> 265

is <vector-name> or <vector-name> ( <a-exp> )

<width> 382

is <deci-con> or <format-var-use>

## Appendix H

## LIST OF COMPLETE ALGORITHMS

Each of the algorithms listed below is given in the text by a flowchart and a MAD routine, unless otherwise indicated as follows:

F indicates that only a flowchart is given

R indicates that only a MAD routine is given

S indicates implementation as a subroutine

C indicates the inclusion of a calling routine


| | Page |
|---|---|
| The solution of linear equations (two versions) | 24-25 |
| The sum of n numbers (two versions) | 29-32 |
| Demonstration of basic input and output (R) | 124-125 |
| Illustration of unconditional transfer | 149 |
| Conversion of numerical grades to letter grades--demonstrating the variable transfer or switch (two versions) | 154-155 |
| Example using simple conditional transfer | 162 |
| Illustration of simple conditional statements--not involving transfer statements | 164-165 |
| Example of the use of a flag | 166 |
| An income tax calculation--demonstrating the compound conditional statement | 178-179 |
| Quadrant of a coordinate point (two versions) | 181-183 |
| Solution of quadratic equations | 186-187 |
| The sum of n numbers--demonstrating flowchart iteration box and MAD iteration statement | 204 |
| A sales tax calculation | 207 |
| Search for the subscript of the largest element of a vector | 209 |
| Search for the first element of vector near a given value | 212 |
| Half interval iteration to root of a function (F) | 226-227 |
| Newton-Raphson iteration to root of a function (F) | 236 |
| Production of a table of square roots | 239 |
| Production of the second through eighth roots of positive integers up to one hundred | 242 |

| | Page |
|---|---|
| Finding all permutations of three things (two versions) | 243-244 |
| Area under a curve by trapezoidal rule and Simpson's rule | 252-253 |
| Approximation to the slope of a curve (R) | 259 |
| Demonstration of the use of the square root subroutine (C) | 268 |
| Demonstration of the use of the complex subroutines (C) | 272-273 |
| Calculation of the real roots of a polynomial (C) | 275 |
| Sorting integers into primes, powers of a prime and others (S,C) | 277, 299-300 |
| Function to calculate the mean of n numbers (S,C) | 284-288 |
| Procedure for complex multiplication (S) | 291 |
| Function to approximate the area under a curve given by a vector of values (S) | 293 |
| Function to approximate the area under a curve given by a function subroutine (S,C) | 294-295 |
| Subroutine to find the roots of a quadratic equation (S,C) | 296-297 |
| Example of an internal subroutine (S,C) | 304 |
| Test for matrix equality (S) | 321 |
| Multiplication of a matrix by a constant (F,S) | 322 |
| Addition of two matrices (R,S) | 322 |
| Multiplication of two matrices (F,S) | 323 |
| Interchanging two rows of an array (S) | 332, 337 |
| Multiplication of a row of an array by a constant (S) | 332, 337 |
| Addition of multiple of one row of an array to another row (R,S) | 337 |
| Pivot for a Gauss-Jordan reduction (S) | 334, 336 |
| Gauss-Jordan reduction | 335-337 |
| Example of the use of a format variable (R) | 395 |
| Example of a recursive subroutine (R,S,C) | 407 |

Index


Abacus, 36, 40
Abbreviations in MAD, 95, 341
Access time, 63
Accumulator, 68
Addresses, 68
Aiken, Howard, 38
Algorithm, 1, 7
  Euclidean, 8
Alphabet, 16, 89
Alphanumeric, 90
Analog computer, 40
Analytic engine, 37
Annotation, flowchart, 11, 15
Arabic notation, 36, 344
Areas under curves, 248
  left sum, 249
  right sum, 249
  Simpson's rule, 250
  trapezoidal rule, 250
Arithmetic expressions, 4, 131-132
Arithmetic operations, 4, 127
Arrays, 96, 309
Assembler, 3, 68, 74
Assembly language, 71, 73
Associative law, 56
Associative memories, 67
Autocoder, 73
Automata, 37, 40
Averaging, 229

Babbage, Charles, 37
Backus Naur form, 88
Backus normal form, 88
Base, of number system, 344
Base element, 109, 309-310
Batch processing, 82
Binary
  codes, 44
  machine, 39, 49
  numbers, 343-360
  point, 356
Binary-coded decimal, 45, 47
  subroutines for, 376
Bi-stable, 38, 44
Bit, 44
Bitwise operations, 141
  produced word, 142
Blank, 16, 18, 89
Boole, George, 38
Boolean mode, 156
Boxes, flowcharting, 12
Branching, 145
Burks, Arthur, 39

Calling list, 266
Carriage control character, 121-122
Character, 16, 87, 89
  field specifications, 383, 394
  oriented machines, 48
Characteristic, 54, 102
Chopping, 54, 57
Codes, binary, 44
  operation, 67
Comments, 18, 119
  printing of, 121
Common storage, 404
Compiler, 3, 73, 74
Complement representation, 51
Complex arithmetic, 373
Conditional statement
  compound, 173
  simple, 21, 160
Connector, flowchart, 15
Constant, 100
  arithmetic, 104
  Boolean, 104
  character, 101
  compilation, 100
  core, 107
  floating point, 102, 104
  integer, 101, 103
  metaconstant, 88
  numeric, 16
  octal, 101
  operand, 100
  subroutine, 264
  truth, 103
  unsigned 101, 103
Content addressed memories, 67
Control, 42, 145

Controller, input-output, 43
Conversion, between number systems, 350
Core constant, 107
Core memory, 63

Data, input, 7, 100, 116, 377, 397-398
Debugging, 361-366
Decimal machine, 49
Decimal point, 100, 356
Decision box, flowchart, 13
Decision table, 71
Decisions, 145
Deck, of cards, 60
Declaration, 16, 94, 409
Derivatives, 256
Diagnostics, 75
Difference engine, 37
Digit, 16, 89, 344
Digital computer, 40
Disk storage, 65
Division, by zero, 131
Double precision arithmetic, 374-375
Drum storage, 65
Dummy name, 279

Eckert, J. Presper, 39
End of file, magnetic tape, 397
End tape, 397
Entry point, 12, 281
Equations, linear, 309
Equivalent storage, 403
Erasable storage, 404
Errors
  execution, 361-365
  fatal, 77
  input and output, 400
Excess codes, 54
Execution, 69, 82, 85
Exit point, 12, 282
Expression
  arithmetic 19, 127, 131-132
  Boolean, 160, 168-171
  floating point, 136
  function name, 265
  integer, 136
  label, 147
  metaexpression, 88-89
Extent, 109

Factorial, 270
Fatal error, 77
Fetching, 69
Field specifications, 379-388
  modification of, 393-394
File, tape, 397
Flag, 163
Flip-flop, 38
Floating point
  field specifications, 384, 393
  machine representation, 54, 57
  notation, 16, 54
  range of values, 102
Flow, of control, 11, 67-69
Flowchart, 11
  iteration box, 200, 205
Formal parameter, 279
Formal parameter table, 280
Format
  expression, 379
  specification, 397
  variable, 395
Formatted input and output, 377-396
Function, 264

Gauss-Jordan reduction, 325
Goldstine, Herman, 39

Half interval method, 218, 269
Hardware, 73
Hindu-Arabic notation, 344
Hollerith, Herman, 38
  cards, 38, 61
  field specifications, 386
Hybrid
  computer 40
  subroutine, 264

Identifier, 96
Increment, loop, 194
Indirect addressing, 72
Initialization, 194, 201
  subroutines, 376
Input, 7, 17, 60, 82, 377, 396-398
  basic, 116
  device, 41, 60
  tape, 82
Instructions, computer, 67
Integer
  constants, 16, 103
  field specifications, 382, 393
  variables, 107
Internal subroutine, 303

Interpretive languages, 74
Inter-record gap, 397
Iteration
  linear, 217
  statement, 199, 205
Iterative
  solutions, 197, 217
  techniques, 197

Jacquard, Joseph, 37
Job, 82

Label, 16, 146
Language, 87
  metalanguage, 88
Leibnitz, Gottfried, 36
Letter, 89
Library, subroutine, 74, 93, 367-376
Line specification, 379
Linear
  equations, 309
  iteration, 217
Linearization, of algorithms, 23
Lines, flowchart, 11-12
List
  actual parameter, 266, 280
  calling, 266
  formal parameter, 280
  operator, 91
  push-down, 405
  subscript, 106
List processing, 405-408
Listing, 60, 82
Load point, 397
Loading, 69, 82
Look ahead, 72
Loop, 26, 193
  index of, 194
  infinite, 150
  initialization, 194
  nesting, 198, 238
  structure, 195-196
  termination, 194-195, 202, 206

Machine, 35, 40
  binary, 49
  characteristics, 50
  character-oriented, 48
  decimal, 49
  language, 2, 67
  word-oriented, 48
Macro, 73
MAD, 3
MAD compiler, 76
Magnetic tape, 66
  input and output, 397
Magnitude, signed, 51
Main routine, 93
Mantissa, 54, 102
Map, 85
Mathematical functions, 21-22, 367-370
Matrices, 319-320
  elementary transformations on, 326
Mauchly, John, 39
Memory, 41, 63
  allocation of, 65
Merge point, flowchart, 15
Metaconstant, 88
Metaexpression, 89
Metalanguage, 88
Metavariable, 88
Mixed orientation machines, 48
Mode, 96
  digits, 98
  indicator, 103
Multiplier-quotient register, 68

Name, 16, 96
  statement label, 146
  subroutine variable, 264
  variable, 106
Negation, 141
Nesting, of loops, 198, 238
Neumann, John von, 39
Newton-Raphson iterative method, 232
Normalization, 54
Null, 91
Numbers, 16, 343-360
Numerical integration, 248, 251

Object language, 74
Octal
  constant, 101
  digit, 101
  field specification, 383, 393
  point, 356
One statement internal function, 303
Operand, 127
Operation
  arithmetic, 127
  binary, 127-129
  Boolean, 168
  box, flowchart, 13

Operation (*Cont.*)
  computer, 67
  fundamental, 8
  unary, 127-128
Operator
  arithmetic, 18, 127
  binary, arithmetic, 127
    bitwise, 142
    Boolean, 169
  relational, 20, 157
  unary, arithmetic, 127
    bitwise, 142
    Boolean, 169
Origination box, flowchart, 14
Output, 60, 82, 378, 397-398
  basic, 116
  block, 120
  device, 41, 61
  list, 120
  tape, 82
Overflow, 130, 371

Parameter, 7
  actual, 265
  formal, 265, 279
  use table, 280
Parity codes, 45
Pascal, Blaise, 36
Pivot, 326
Polynomial evaluation, 211-213
Positional notation, for numbers, 344
Powers, James, 38
Precedence, of operators, 133, 143, 170
Predefined process, flowchart, 14
Printing, 16, 61
Problem oriented language, 73
Procedure, 264
  elementary, 8
  languages, 73
Process, 4
Processing, 13
Processor, 41-43, 67-72
Program, 11, 93
  computer, 11
  stored, 39
  system, 74
Punched card, 60
Push-down stack, 72,405

Quadratic equations, 185

Radix, of number system, 345
  point, 356
Random-access, 63
Random number generation, 372
Range, of subscripts, 313
Real-time computation, 62
Record, tape, 397
Recursive subroutines, 405, 407
Recursiveness, 90
Redundant codes, 45, 46
Registers, computer, 67-68
Regula falsi, 218, 228
Relation, 20, 156-157
Remark, 94
Remote consoles, 43, 83
Repetitive processes, 191
Results, 7, 119, 127
  printing of, 120
Return, from subroutine, 282
Rounding, 54, 57
Routine, 11, 93
  main, 11
Run, computer, 82

Scaling, 53
Scientific notation, 54
Scope, 200, 205
Search, 198
Sign, 100
Signed magnitude, 51
Simultaneous linear equations, 325-326
Skip field specifications, 382
Slide rule, 40
Slope of curves, 256
Software, 73
Source language, 74
Span, of subscripts, 313
Statement, 16, 94
  assignment, 19, 127, 139
  conditional
    compound, 173
    simple 21, 160
  declaration, 94
  executable, 94
  label, 146
  remark, 94
  substitution, 19, 127, 139
  transfer, 20, 148, 152
Step, in loop, 194
Storage, 41, 63
Stored program, 39

Storing, 69
String
  character, 112
  operator, 91
Structure, 109, 312-315
Subroutine, 11, 93, 261
  box, flowchart, 14
  calls, 263, 266
  error condition, 267
  external, 262, 279
  internal, 262, 303
  tracing, 365, 375
Subscript list, 106
Substitution
  statement
    arithmetic, 4, 139
    Boolean, 168, 171
    function name, 265
    label, 147
  box, flowchart, 13
Symbol, 89
Syntax, 87
System
  computer, 75
  program, 74
Switch, 151

Tabulation specification, 387
Tape, magnetic, 66, 396-398
Tattle tale, 89
Termination
  box, flowchart, 14
  of iteration statements, 202, 205
  of loops, 194
Through-put, 81
Time sharing, 83
Transfer
  of control, 20, 69, 145
  unconditional, 148
  variable, 151
Transformations, elementary matrix, 326
Translation, 74
Truth Table, 141, 168
Turing, A. M., 38
Turn-around time, 81

Underflow, 130, 371

Variable, 106-107
  data, 116
  format, 395
  input, 377
  output, 378
  statement label, 146
  subroutine, 264
Vaucanson, Jacques de, 37
Vector, 17, 109
  initialization, 112, 114
  name, 111
  base, 111
  block, 112
Verification, of algorithm, 9

Word, 87
  oriented machine, 48

Zero, division by, 131

## RELATED TEXTS FROM MACMILLAN

**FORTRAN II and IV for Engineers and Scientists**

*By Hellmut Golde, University of Washington*

This basic, freshman textbook provides a parallel treatment of FORTRAN II and IV with the primary objective of teaching students to use digital computers early in their course work. It may be adapted, with the aid of Appendices A and B, to practically all FORTRAN systems. FORTRAN is presented within the context of typical computing problems taken from various areas of science and engineering and many illustrative problems are presented. Sample problems are graded in degree of difficulty as the student progresses from chapter to chapter. All programs are original. 224 pages.

**Introductory Computer Methods and Numerical Analysis**

*By Ralph H. Pennington, Air Force Weapons Laboratory*

This book provides a well-organized development of computer methods for solving the more elementary problems in applied mathematics. It also presents an advanced discussion of techniques aimed at giving the student a perspective on the relative difficulty of problem-solving on a computer. Designed for one-year freshman and sophomore courses, the book is addressed to the student who has an understanding of mathematics through integral calculus, but little or no familiarity with the digital computer. 452 pages.

**Computer Methods In Solid Mechanics**

*By Joseph J. Gennaro, Stevens Institute of Technology*

*Macmillan Series in Civil Engineering*

This volume develops the fundamental methods of structural analysis, using matrix terminology and paralleling basic programs which permit use of a digital computer. The programs presented are written in FORTRAN and have been fully tested on an IBM 1620 computer with 40K memory. They are developed in a perfectly general manner so that they may be considered analogous to new mathematical methods of analysis. Punched data cards are available from the author.

292 pages.

THE MACMILLAN COMPANY
866 Third Avenue, New York 10022

37589